Tools and processes for creating and maintaining own Linux based distributions in corporate environment

Juhani Heliö

Helsinki April 28, 2017

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Juhani Heliö

Työn nimi — Arbetets titel — Title

Tools and processes for creating and maintaining own Linux based distributions in corporate environment

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	April 28, 2017	70 pages + 17 appendices

Tiivistelmä — Referat — Abstract

Nokia has been maintaining its own Linux distribution, dubbed Nokia Linux, for quite some time, and while the distribution has been successful, supporting it has become tedious at very least. The current version of Nokia Linuxes build system builds a monolithic distribution. This creates a multitude of problems ranging from having to rebuild all of the distribution's software packages after patching a single package to not being able to efficiently customise images according to user needs. The current version also lacks any kind of stable release management causing every user to either maintain their own stable releases or having to constantly update from the official version. Apart from being too monolithic, the support of the software packages is insufficient. An efficient support team needs to be created to answer any internal support requests Nokia Linux users might have.

In this thesis we first define our corporate environmental needs. We identify three needs: support, storage and security. We then define three methods for organising the support. We compare them then to the current way of delivering support. We conclude that the three methods are probably better than the current one, but more research is needed before any definite answer can be given.

We then define three processes for creating and maintaining an own Linux distribution in our corporate environment. We show two of these processes in action through a proof of concept project. With this project we show that the two processes we tested can be used to create and update an own Linux based distribution image composed of RPM packages. These two processes were tested with multiple tools to show that these processes can be used flexibly and without having to lock into only one set of tools.

ACM Computing Classification System (CCS 2012):

- **General and reference ~Design**
- **Social and professional topics ~Systems planning**
- **Social and professional topics ~Systems analysis and design**
- **Social and professional topics ~Software selection and adaptation**
- *Social and professional topics ~Project staffing*

Avainsanat — Nyckelord — Keywords
Design-science, Linux, Kiwi, Koji, IT Support team, Linux build system, Corporate environment

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

From its humble beginnings Linux and its various distributions have spread across the IT world to become the top operating system for software development, big data management and hobbyists alike. At the time of writing, Linux keeps the Internet up and running for our convenience. Linux based operating systems, like Android, are the most popular operating systems in mobile devices [GRT]. On the field of supercomputing, Linux is the de facto operating system being in use with over 99% of the top 500 supercomputers using Linux based operating systems [TOP]. At the embedded and real-time operating system side of the field, Linux is dominating with a share of 56% of devices shipped worldwide with a free Linux as the operating system in 2012 [LEm]. The next largest group in this category only has a share of 30% [LEm]. In 2016, the amount of embedded and real-time devices with Linux as the operating system, is expected to rise up to 65% [LEm]. From the independent software project that begun in the early 90s, Linux has grown to be the largest open source project in the world and is not showing any signs of stopping any time soon [Ray99].

The success of Linux stems from its open source nature [Ray99]. Anyone could create their own Linux distribution and distribute it to the world. This gives companies a possibility of tailoring a free operating system to their needs. However, if a company were to create a distribution tailored to its needs, the new distribution would then need to be maintained. If this distribution is used in the products of the company, the maintenance of the distribution would then have to be done by the company to prevent trade secrets from leaking.Although the source code of the underlying Linux is open to everyone, the product components running on top of that Linux might not be so open. The maintenance of the company distribution would for example require the company to maintain its own software package repository to ensure independence of the package provider. Security would be a high concern and stability a requirement. The distribution could be needed to be built in a flexible manner and the building process be automated to ensure consistency of quality of the distributions.

The goal of this thesis is to study how the creation and maintenance processes of the Nokia Linux can be improved and made more flexible compared to the current ones. These improved processes should also try to minimise the need for users inside Nokia to maintain their own versions of the Nokia Linux.

1.1 Nokia Linux

Nokia has been using and maintaining its own Linux distribution for some time now. Its current iteration, dubbed Nokia Linux, was first created in 2013 and has seen widespread use inside Nokia. This success, however, comes with a drawback. Due to so many users, the maintenance of the distribution for various needs inside Nokia has become tedious. Identified major problems with the current version of the distribution are lacking package management and package customisability, technical support team and stable releases. Without solutions to these problems, maintenance of Nokia Linux is feared to become a costly task, and a task that could slide to production teams to manage.

Nokia Linux distributions are monolithic in nature. This creates a problem when maintainers of Nokia Linux would want to apply new patches or install new software. The monolithic nature of Nokia Linux comes from how the build system of Nokia Linux, Hammer [AKH], builds the distribution. Hammer uses source packages taken directly from their upstream repositories, compiles them, and installs them to the system using a set of custom build scripts. These scripts compile and install all the packages in one go and cannot be ran separately. When a user would want to install or modify a piece of software, it has to be done using Hammer. If one package would receive a patch, in order for that patched version to be incorporated, every other package would need to be recompiled and reinstalled as that is the only way Hammer can install or modify packages. Even with a more powerful computer it can take longer than one and a half hours for the build to finish. As the current Nokia Linux distribution is over 1 GB in size, rebuilding, sending it to users, and users installing it to their machines, can take considerable time and effort.

To address this problem, the next iteration of Nokia Linux is considering adopting a more flexible build process. However, as Hammer is a similar Linux build system as the Linux from scratch project, we would need to modify Hammer to accommodate the new build process. Although this possibility might not be as insurmountable, we would rather look for an already existing alternative.

I, with the help of other Nokia engineers, have studied different Linux build systems that would suit our needs. Four different build systems were chosen for this thesis: Fedora's Koji Project, openSUSE's Kiwi, Yocto Project by a open source collaboration team led by Linux foundation, and OpenStack's Diskimage-builder. Out of these four we chose two, Koji and Kiwi, to be tested with the processes defined later in this thesis. To complement these, we looked into rpmbuild and Mock for building

binary packages, and createrepo and Mash for creating package repositories.

As the Hammer builds all the packages from sources, installs them with custom scripts, and has no efficient package manager, the build system itself cannot build the packages out of some list. It instead requires the source code of the software. The source code of the software being used is taken from the authors repositories directly. Mainly these are individual git or similar repositories. This means that in order to customise the package composition of Nokia Linux, users would have to provide the source code and then modify the Hammer build scripts to accommodate this new software.

Also, the Nokia Linux distribution has to include all the packages required by all the users even though most of the users would not benefit from all of those packages. If users do not want to have some package in their systems, they have to maintain their own fork of the current Nokia Linux and do the modifications themselves. This is not the optimal solution. If users would start maintaining their own forks, users would need to use time and effort on fork maintenance instead of their own work. The new build processes should be able to create new Nokia Linux distribution images with custom package compositions without pushing the maintenance to the users.

Current Nokia Linux has no support team to assist with different packages. This is a major inconvenience for users who run into problems with, for example, usage of a new feature of some software. In the case of our proof of concept where we would have a package manager incorporated in our Nokia Linux, users could also face problems, for example, when installing some new piece of software. In these example cases and without a support team to help, users would have to search for the solutions by themselves. Package authors and upstream communities could be leveraged for help but these might be unresponsive or even dismissive.

Currently, if users wanted some customisation on their Nokia Linux distribution image, they would have to maintain the distribution image themselves. This would mean that they would effectively become the support for their own distribution. If we want to have a system that would be more automated, we would need to centralise the package support as a support team. This team should be able to perform as a stable line of support for anyone using Nokia Linux. The support team or teams would also be tasked with upgrading packages and handling security issues, to name a few responsibilities. In this thesis we have discussed about how to make the support as accessible to users as possible. The responsibilities of this support

team have already been evaluated internally at Nokia. We will touch some of these responsibilities in our study as well but it will not be a primary objective of the support team study. We have created three new proposals for how support teams could be organised.

Conscious architectural decisions are the main cause of the problems plaguing the current Nokia Linux implementation. At the time of first creating the Nokia Linux these decisions were sound as the current iteration of Nokia Linux began as an operating system for a single product. As the Nokia Linux began to be used by more and more products and teams, the distributions of Nokia Linux began to lose focus. To avoid the current Nokia Linux to become hard to maintain, this thesis has studied different approaches to replace the current build system and process with a more flexible one.

1.2 Research question

RQ1:What are the corporate requirements we have to consider?
RQ2:How could the support teams of our Linux distribution be organised to increase the availability of support to the users of our Linux distribution?
RQ3:What are the processes for creating and maintaining our Linux distributions?

1.3 Methods

In this thesis we used design-science research methods, described by Hevner et al. [VAMPR04], to construct two sets of artefacts. The artefacts were evaluated using experimental and descriptive studies. We aimed to produce a set of artefacts for creating and maintaining corporate owned Linux distributions. These include a set of processes for using tools to create and maintain a corporate owned Linux as well as a set methods for organising support teams.

We analyse the processes for using tools through an experimental study in the form of a proof of concept. This study was done in our corporate environment. We used different tools for testing the processes to provide proof of the validity of the flexibility of our processes.

We provided propositions on how the support teams for this new Linux distribution could be organised. These methods were proposed as suggestions and a descriptive

study was done to give some informed arguments and to create scenarios for these methods. We draw conclusions in the form of a scenario on how the different methods could be used in different situations.

We feel that the artefacts constructed in this thesis contribute to the design-science research in the form of a case study. This case study provides an example on how corporate environment can affect Linux distribution creation and maintenance process' design.

1.4 Structure of this thesis

In this Chapter, we went through some of the key problems of our current Nokia Linux. The rest of this thesis can be divided roughly into two parts: The first part will be exploring key concepts required by the second part. The second part will then create a proof-of-concept solution for our processes utilising the study done in the first part. The final conclusions will be given at the end of the second part.

In Chapter 2 we will take a more in-depth look into some of our key corporate environmental factors that we need to accommodate into our work. We have identified three factors: Support, Storage and Security. Chapter 3 will be exploring the first one of these, the support. More specifically, how support could be made accessible to the users. In Chapter 4 we study the second and third of the key corporate environmental factors: Storage and security. These two key factors have been the focus of our study about how to create binary RPM packages as well as how to store them.

In Chapter 5 we will begin our proof of concept. In Chapter 5 we define three processes: creation of a new Linux distribution, updating an existing Linux distribution and creating a new release out of an existing Linux distribution. Chapter 6 devotes itself to the exploration of the tools to be used in our proof of concept. We have 6 tools we will be using in our proof of concept: rpmbuild, our Koji servers Mock, createrepo, Mash, Kiwi and Koji. In addition we also briefly look at alternatives like the Yocto Project and OpenStacks Diskimage-builder. In Chapter 7 we will present our test setup for our proof of concept. We have four test setups that will be used in our proof of concept.

In Chapter 8 we go through our results from the proof of concept. These results are then discussed in Chapter 9. Source file listings are given as appendix.

2 Corporate environment

Corporate environment is quite a vague term. It can be used to describe many aspects of corporate life, like the actual physical environment at an office or the sociological environment created by the employees. Many things affect corporations and the different environments within. For example, size, in terms of how many employees and how influential the corporation is, has an effect on the nature of corporation environment. As the size of a company increases, the amount of organisations inside that company tend to grow as well. More people need to be organised into small enough organisations, like teams for example, to enable the effective use of the people in the company. For example, large teams would most likely have too many employees working on a single project. Some of those employees could have very little or nothing to do and would be, figuratively speaking, wasted potential.

For our purposes, we use the term ”corporate environment” to refer to a collection of organisations that make up a corporation. Inside larger companies, there can be dozens of organisations, big and small. These organisations can be anything from product lines to single products and have sub-organisations (individual teams, for example). In this thesis we have studied how this kind of corporate environment can be satisfied with a common Linux distribution. The goal of this thesis is to find a more flexible way of creating an operating system for these kinds of corporational needs.

This all affects the creation of an operating system by imposing some restrictions and giving some requirements what the operating system should do. For example, in the case of company size, if the company is large, the operating system should be able to support the work done in many different production lines. Otherwise, the production lines would have to create and maintain their own operating systems. If this would be the case, then some parts of the operating systems would be the same or very similar across all the different operating systems. It would be wasteful to maintain the same thing in many places at the same time. This problem could be solved by creating a modular operating system using software packages. These packages have been the staple of software delivery for Linux for quite some time. We take a closer look at the software packages in Chapter 4 of this thesis.

In this Chapter we have focused on the requirements imposed by the corporate environment. However, as there are different corporation environments, there are also different requirements imposed by these environments. In this thesis we are

not trying to present a comprehensive list of different requirements. Instead we are going to define them from the perspective of the corporate requirements of Nokia. We have explored the operating system building from the point of view of three key requirements. These three key requirements have been identified as key requirements internally at Nokia. The three requirements are support, storage and security.

2.1 Support

From the operating system point of view, having a large number of organisations inside a corporation is going to make the maintenance of the operating system for all the organisations tricky. The same operating system will not work for everyone as, for example, an organisation might want a package A that would then conflict with package B required by another organisation. An efficient means of maintaining and support not only these packages but the distributions containing these packages, the distribution and package build tools and the processes for using these, must be created in order for the operating system to be feasible to maintain. If this kind of maintenance would not be available, organisations would need to maintain their own Linux distributions. This, as discussed earlier in this thesis, is not desirable.

What we propose in this thesis is an organisational package management model for corporations. We would have a core package repository that would contain all the packages that would be required by every organisation. Organisations themselves would maintain their own repositories for organisation specific packages. This would have the benefit of sharing the load between organisations. This would also ensure that the required skills are coupled to the packages. As the organisations would be hosting the packages they need, they would also have personnel capable of understanding the packages as well as the reason of having said packages.

2.2 Storage

For any corporation, efficient utilisation of software packages will require collecting those software packages which are meant for some particular need, to a common repository. This enables the users of these packages to use minimal effort to find the packages the users would want to find. Especially when utilising a package manager, if the packages are in some compatible repository, the package manager finds the packages the users need automatically. If the packages would be scattered in many repositories, users would need to know the locations of these repositories,

and whether the repository would actually contain the needed packages. These repositories would then need to be configured so that the package manager can use the repository. When using only a couple of repositories, this effort is minimised.

Good examples of how the packages could be distributed between different repositories would be to do, for example, what Fedora and openSUSE are doing. Both of these have, for example, the updates and base packages in separate repositories [FeP][OPR]. This means that support teams, and everyone else using these packages, can more easily keep track of the whereabouts of these packages and make a distinction between base packages and the updates to those packages. If the packages and their updates were scattered into too many locations, users of these packages would have to keep a long list of locations, and even then finding the right location would require some effort. This would increase the maintenance effort of the distribution as packages would have to be fetched from many different locations that would first have to be configured by the support teams.

Corporations could have a need for their own repositories. Even if the piece of software would consists of only open and free source, there are reasons to have an internal package repository. We have two examples of why corporations should have a internal repository: package availability and independence.

2.2.1 Availability

Corporate environment enforces the need for structured and organised repositories for the software. As the amount of packages utilised by all the organisations can be large, keeping the packages in clearly defined repositories is a necessity. These package repositories also need to be easily accessible for all users. Ease of search is also required as new users need to know the contents of these repositories or the whereabouts of any single package they would need. These repositories need to be developed to ensure high availability and ease of search [DG01].

If the packages would be scattered in many repositories that are not very clearly defined or are, for example, behind some proxy, maintainers would have to deal with many repositories at the same time. With packages being scattered in a large number of different repositories, problems that may arise include having the same package present in multiple repositories and increases the risk of having dependency problems between packages originating from different repositories.

2.2.2 Independence

A key role of the package repository would be to provide all-time availability of the packages to the company regardless of the official software package repositories. With this, we mean that the contents of a repository should be accessible all the time. In the case that the common, public package repository containing some package a company needs goes down or would be temporarily inaccessible, the company should still be able to continue its work without major interruption.

Also, corporations could want to have some custom versions of the open source packages. Some versions could for example have changes that were not approved to the official repositories but are still desired by the corporation. Also modifications to the packages that could be integrated to the upstream packages, but are still being considered by the upstream, could be stored here.

2.3 Security

Corporations guard their secrets very closely. This is also a reason for the need of creating independent repositories for packages. Creating these repositories enables corporations to make sure the components used in, for example, the corporation specific Linux distributions, are free of security faults. If a fault would be discovered in a corporate internal review of the stored software, a fix could be immediately applied and the author of the package could be notified. When using public repositories, in a similar situation, the author would be notified first and a possible fix proposed, but applying the fix would be left for the author. Author could change the fix in a way that would still leave some vulnerability from the company's point of view or even discard the proposed fix. This would force the use of other means to apply the fix, such as creating a fork of the package that would be then maintained by the corporation.

Corporations would also need to be sure that the software they are using is malware free. No company wants to have a security vulnerability in their systems due to a bug in some open source package. Even worse, careless use of software, proprietary or open source, could even create a security vulnerability in some of the company's customer's system causing possibly large scale harm to both the providing company and the customer. This is also one reason for having company's own repositories of packages. We can then inspect the packages and ensure that there are no security vulnerabilities.

With the packages in our own repositories we can also make sure that there is no intentional tampering of the software. Some malicious parties would want to exploit a company's usage of open source packages by intentionally inserting malicious code. We also want to make sure our packages are free of this kind of tampering.

Despite the scenarios described above, open source software is still found to be as secure as proprietary software. For example, in his work, Schryen found that there has been no significant differences between FOSS and propriety software security issues [Sch11].

3 Support teams and support accessibility

As Linux operating systems are built with a large number of packages, giving meaningful support will be a task that will not be feasible for a single person. For this we need a team to be able to give support for all the packages in a distribution. For example, Nokia Linux has currently about 200 core packages, not counting the development packages needed in some builds. For a single person or a small team, supporting this number of packages would be an impractical task. The lack of support for the packages in the current Nokia Linux has been identified internally by Nokia to be a major problem. No official team exists that could give detailed support in questions related to the packages of the Nokia Linux. This means that users have to use time and resources to learn what the packages do or turn to the upstream communities and authors for support when facing problems. This might take considerable time and resources as interacting with upstream communities might not always be easy. For example with smaller, less active communities the response time for questions might be long.

Of course, there are lots of employees at Nokia who have intimate knowledge about the functionalities and problems of some of the packages of Nokia Linux but these people are not organised in any way. These employees might even be active contributors in some open source communities. These knowledgeable employees would have to be found through what is basically hearsay. If a suitable employee was found, the employee might be too busy with their own work to be able to give any meaningful support.

One of the requirements imposed by the corporate environment is the need for efficient support for the packages, build process and tools, and the distribution itself. In this Chapter we have explored the possibility of having a support team capable

of accomplishing this task. We have first looked at some of the responsibilities identified by an internal study. We will extend the findings of this internal study by exploring how the support teams could be organised to make them accessible to the users. This support team should be organised in such a way that people in need of support would be able to get it as easily and conveniently as possible. For this, we propose three models for organising the support. These support models ponder on how the granularity of the support team or teams would affect the availability and quality of the support given by the team or teams.

3.1 Responsibilities of teams

The current iteration of Nokia Linux has about 200 packages that would need to be supported. This means that the Nokia Linux support team would have to have good knowledge on how the packages work by themselves and in concert with each other. It is important to remember that the packages are not isolated things inside the operating system but rather depend on each other. This package interdependency can introduce weird and seemingly random errors and bugs that the support team should be able to handle. Because the amount of packages in an operating system, be it ours or some others, is large, no single person is expected to have complete knowledge of all the packages and their behaviour together. For this we need a team of sufficient size to be able to distribute the load.

If an in-house support organisation would be created it would need to be able to handle almost every kind of question about the packages and their technical details. If problems would arise with, for example, installation, the in-house support should be able to respond and resolve the situation. The teams should be able to help with technical problems and functional details of the packages. The support teams should be easily accessible and in the case of finding singular support persons or teams a clear list for the teams and persons with their roles and competencies should be shown. This would help make asking for support more accessible. Users could search this list easily and find the person they would need.

One of the key aspects of the support team would be to act as an interface between the open source communities and Nokia's developers. Currently one of the major problems of Nokia with open source communities is the lack of experience with communication. There have been cases where a developer has tried to contribute to an open source project but because of poor communication this has been ignored or even denied. These are not singular cases so an action is required to create bet-

ter communication processes for Nokia to use with open source communities. The support team should therefore have good understanding about the open source communities they are dealing with. One way to achieve this is to have the support team integrate with the communities. This should come naturally as while the support teams would learn about the packages, they would interact with the communities. Indeed, it should be required that the support personnel are contributing at least to conversations ongoing in the communities. This way the personnel would gain reputation inside the communities and gain authority this way. Later the support personnel could even become the maintainers of the entire package like how Raymond gained his project [Ray99].

Should some of the packages need a new feature requested from Nokia or a patch that cannot be readily incorporated into the upstream project, the support team needs to be able to support a Nokia specific fork of the project. This fork could be temporary as to allow the Nokia developers to use the desired feature or patch while it would be integrated to the upstream. Bear in mind that this integration could take days or even weeks as there are legal checks, for example. If the project would be very inactive, it could be possible that the fork could become the project itself. If the project is no longer maintained by anyone or if the maintenance is not frequent, the project could be adopted and the fork could become the project.

These requirements for the teams mean that individuals in the teams must be very focused on the package or packages they would be supporting. In this thesis we have studied the efforts required to organise support teams in such a way to best support the development teams. The second objective of our support team study was how to make team members integrate themselves within the open source communities.

3.2 Integrating support teams with open source communities

In order to get the support personnel to become experts in the packages they would be supporting, they could be assigned to take active part in the communities of the packages they would be supporting. This would mean that the personnel would be able to know the current administrators and maintainers as well as key contributors. The support personnel would be then able to assist developers on issues related to the packages. This way quality assurance could be controlled inside the company employing the support personnel as the support personnel could observe the quality of the software and make modifications accordingly.

This training, however, takes time and could be prone to risks. If the support trainee is not accepted into the open source community or if the community is very inactive, it can be impossible or infeasible for the trainee to become a fully fledged expert in the particular package. In both cases, however, one could argue that forking the project could resolve this issue but then extra effort would have to be to, for example, keep up to date with the official version of the forked software.

Becoming a member of an open source community has been studied more by Jensen et al. [JKK]. In their study, Jensen et al. go through the phases of becoming a working member of an open source community as well as some of the roles of the people in the community. Another example of interacting with open source communities comes from Raymond [Ray99]. He recounts his experiences of first becoming a contributor and then inheriting an entire project in his book "Cathedral and Bazaar".

3.3 Importance of camaraderie

Most of the large projects are team efforts. Projects that seem on the outside to have only a single creator, most likely have people backing the creator of the project, providing help, for example, in technical and non-technical questions. The more the backers involve themselves in the project with the creator, the more the backers and the creator form a sense of common purpose and mutual understanding. This is what is called camaraderie. In this thesis we argue that by utilising camaraderie in support team formation we can make support teams work more efficiently to resolve users requests. We argue that by having people a user knows in the support team makes the support team understand the users request easier, user can be more forthcoming and feel more comfortable as the support team and the user are already acquainted with how each other behaves. However, this is presented here merely as an idea to be studied further and is not studied further in this thesis.

3.4 Commercial support vs company internal support

Many open source groups and companies, such as Red Hat [RHS] and Zimbra [ZSP], are monetising their open source software by selling support services. This means that even though the software itself is free for users, the support for the software would cost the user. Users would either need to learn by themselves or pay thus expending either time or money.

By paying for the support, users and companies could save time. If a project would be time sensitive this could be a worthwhile investment. Otherwise, paying for the support means that the company would not necessarily get the knowledge to resolve a similar case, should one arise again. This contradiction can be thus seen the following way: paying for the support helps a project or a company in the short term to resolve issues. If the need for support would be small this could be a good alternative, as otherwise teaching the staff of a company to be able to handle support situations could in itself be very costly.

On the other hand, if the project or company has a longer term need for support for a certain piece of software, training company staff could be more worthwhile. Training staff for support roles helps to bring expertise of the software into the company's pool of knowledge. Thus, further support requests would not be directed outside of the company, possibly creating savings not only in money but time as well. It could be faster to contact company internal support staff for support rather than reach for outside help. Also, company staff would not have to go through legal proceedings for non-disclosure and would already be familiar with the company's environment. Furthermore, if the support would be local and readily available to the users of the support, the software developers, we argue that the camaraderie between the support personnel and the developers could form or already exist. This could help the company's support staff to understand the support question faster than paid, company external support.

We want to invest in the long term. Because of this, we explore the potential of creating our own support teams and training our own support personnel. This creates the opportunity for us to expand and consolidate our existing knowledge pool and possibly save developer's time. Also, as the support for some of the open source software is handled by the community or authors instead of a dedicated support staff, we would like to have a support staff of our own that would be able to communicate efficiently with these communities. However, paid support could be used to complement the following models. Before we have a fully operational support, we could employ commercial support for those needs we still cannot support. The goal, however, would be to have a full in-house support for the software we are using.

3.5 Proposed methods of organising support for Nokia

In this thesis we have proposed the following methods to use for alleviating Nokia's problem of support availability: The current method of relying on open source

communities and random employees at Nokia for support, to create a single large dedicated team to manage and give support for all the packages of the Nokia Linux, to create smaller dedicated teams for different product needs to support the product specific packages and to create dedicated positions inside teams for support of packages the team uses in their work. Out of these the first one is used as the baseline and will be used to compare all the others against. This will be referred in this thesis as the ”baseline method” in the context of support teams. The second will be referred as the ”general support team method”, third as the ”organisational support team method” and the fourth as the ”distributed support method”. However, due to time and resource constraints, these proposed methods of arranging support are currently purely theoretical. We do not have sufficient time or resources allocated for this thesis work to be able to test these in real life. A separate study should be made to see how real world challenges would affect the effectiveness of each proposed method.

3.5.1 Baseline method

Figure 1 shows the layout of the baseline method. In the current method there is no layers between the developers and the open source communities and if the users would want to have any support, they have to ask directly from the communities.

This method is the current way of working for package support. In this method, the support of the packages is left for the open source or upstream communities and authors. Some of the Nokia employees can be used for support but this group is not coherent and has no common channel for support related questions. The Nokia employees with the knowledge about certain packages cannot be coordinated as there is no list to check who knows what. As no list of package-competent employees exist, a person trying to get support has to find these employees by themselves and often by just knowing the right person who knows the next right person. The employees are also likely to be bound to other duties and will not necessarily have the time to respond to support requests. We rule this kind of random employee support to be too random and sporadic to be of use for the baseline method. Also this kind of random employee support can be included in all methods so it is also a common denominator and can be ruled out. For evaluating the baseline method, we have only studied the current method for getting information from open source communities and upstreams.

Using open source communities and authors for support can be unpredictable. Some

large open source projects like Linux kernel are well supported and the community is large [Lin]. This means that for a support related question, there is bound to be someone with the knowledge. As noted by Raymond, ”Given a large enough beta-tester and co-developer base, almost every problem will be characterised quickly and the fix obvious to someone” [Ray99]. This was dubbed the ”Linus's law” by Raymond. The complement of this would then be that given insufficient number of beta-testers and co-developers some problems are not found and will be non-trivial to fix. Projects with low amount of developers and contributors are going to have bugs and problems that have been missed. These misses can be quite devastating as seen, for example, with the OpenSSL Heartbleed incident that affected a considerable portion of all the Internet servers and clients [HaB].

Another problem of purely using open source communities and authors for support is that the communities and authors are not obliged to provide any support. This means that a user can be ignored and the support request forgotten, and if there would be a security vulnerability caused by one of the open source packages, the community would not be legally responsible for any losses. This applies also to contributing to an open source project. No matter how good the change might be the author or authors of the project can just ignore the aspiring contributor. Larger communities, due to the amount of people, could be easier to get support for problems. Deriving from the Raymond's ”Linus's law” we can state: given a large enough community, there will be someone willing to help a user requesting for support and has the competence to do it. Complement of this is then that given a small enough community some user requests for support are ignored either due to lack of interest, knowledge or inactivity. However, giving support to open source users can also become a source of income for a open source companies. For example, some of the larger open source communities, like Red Hat, actually create revenue by selling support services for users.

3.5.2 General support team method

The main idea would be to have a single big team that could support all the packages of the Nokia Linux. This way the support would be centralised, coherent and readily available in one place. This is illustrated in Figure 2. Also the support team would then be more unified as they would be able to know each other. In the paper ”Studying teamwork in global IT support” Davidson and Tay state that

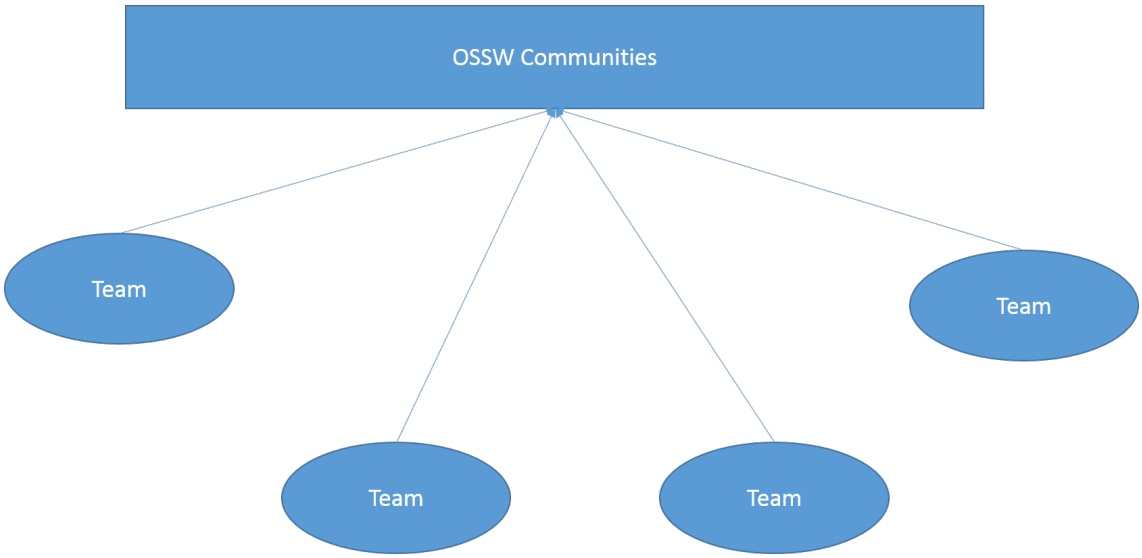

Figure 1: baseline method

”When coworkers are not located in the same location, camaraderie and socialising – important informal aspects of teamwork coordination – are significantly reduced, and cohesiveness and team unity, and the means of socialising new members of the global team, are harder to cultivate” [DT03]. This can lead to loss of efficiency of work as, for example, overheads for communication grow and quality of communication drops. Davidson and Tay also raise this as one of the issues of geographical separation of teams.

This camaraderie can be turned into a disadvantage though. As the support team gets along well with each other, it can lead to alienation from the users. For example, this means that the support team does not have the same kind of vision to the product that is requesting support. This can then lead to misunderstandings and even dismissals of support requests on the support teams behalf as the support teams might not understand why some requests are important. Even if the explanation for the support request might be a sound one the support team might still see that there is no problem.

This is actually partially the reason for creating this thesis. There actually exists a very small support team for Nokia Linux. However, this support team is unable to do any meaningful support of the packages themselves. There have been many cases that the support team for Nokia Linux has said that for the added packages, stable releases or other such examples, it would be better for the teams to form own forks of the current version. These are good points from the support teams perspective

as it most likely would not be feasible for a small team to even try and maintain all the different versions of the Nokia Linux.

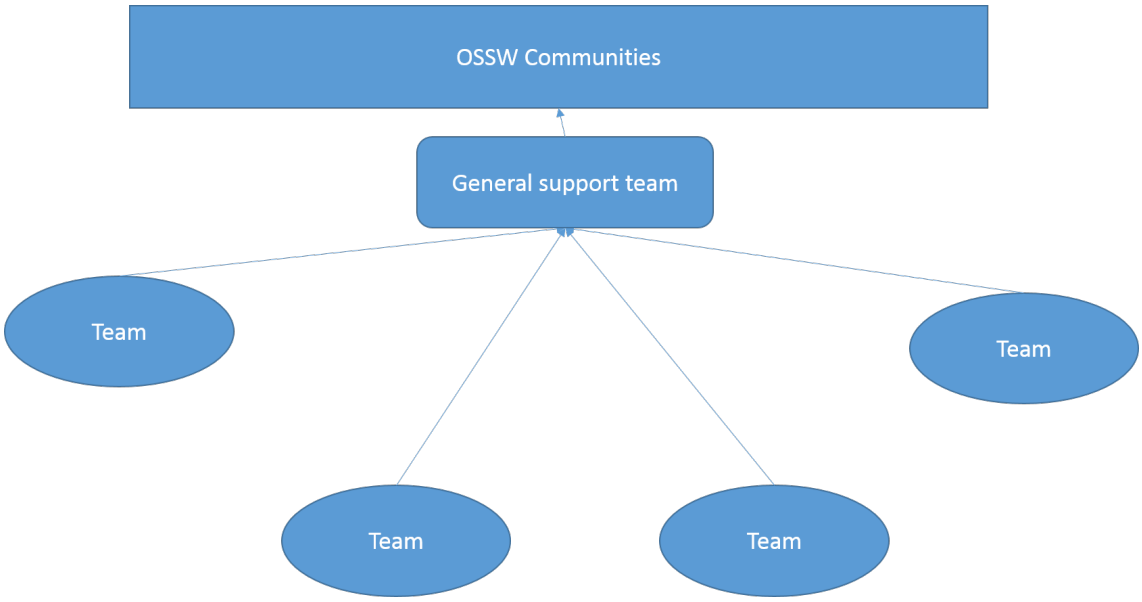

Figure 2: General support team method

3.5.3 Product support team method

In the third method, there is no single large support team, but a group of smaller teams. Every product would have their own support team. This would bring the benefit of having a support team that knows about the product and the packages it uses. This way the support might be able to respond to different questions easier as the support personnel would know the backgrounds of the product. This method tries to bridge the gap between the teams and support by forming the support into layers of teams. An illustration of this method is given in Figure 3.

In this method there would not only be a team to support each product in their specific needs, but also a general support team. However, this general support team would be smaller than in the General support team method. The purpose of this team would be to oversee the support of common packages; packages that every product team needs. Otherwise there could be a large number of overlap between the different teams as most likely, most of the packages would be the same between products. At the very least, most of the base Linux packages would be the same.

The support teams would communicate with each other and with the open source communities and authors for any support related questions that would be out of their area of expertise.

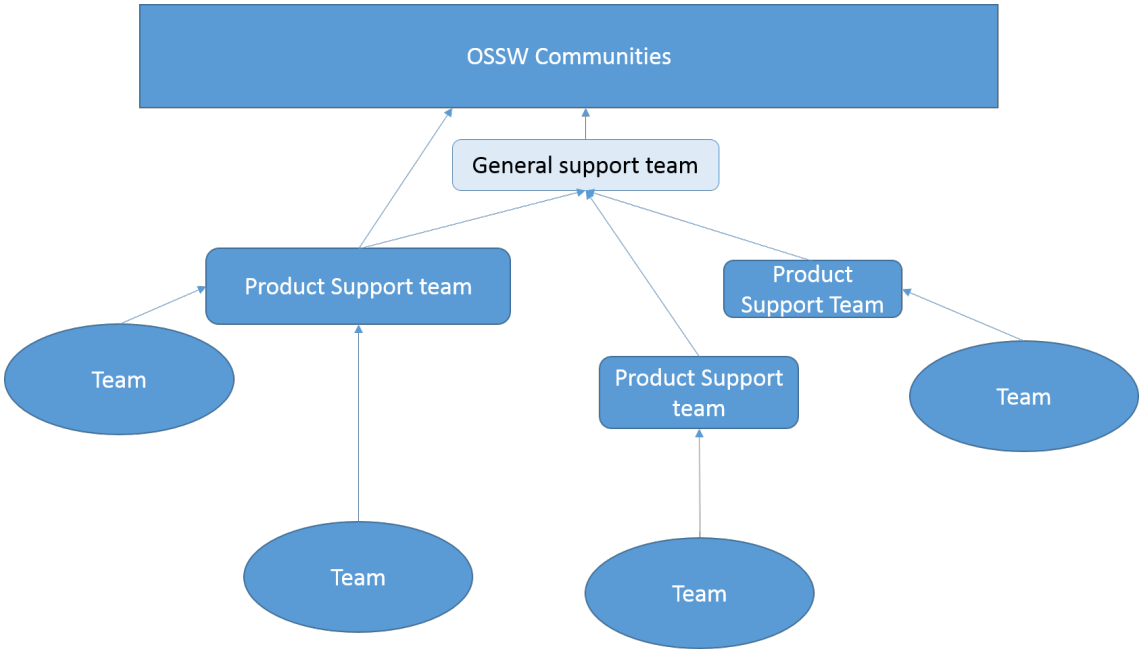

Figure 3: Product support team method

3.5.4 Distributed support method

In this method the supporting personnel would be embedded into the teams. Every team or a couple of teams would have their own support personnel. This would bring the support very close to the teams and the work they are doing, thus reducing the response times. This closeness is illustrated in Figure 4. Also the support could be of better quality when the person requesting support could just walk up to the support person and ask personally. This kind of teamwork and camaraderie is also noted by Davidson and Tay [DT03] to be beneficial.

When using this model, it would be important to have a clear, easy to access and easy to use platform where all the support personnel could be found and their expertise would be easily understood. This would enable everyone to request support from every support team.

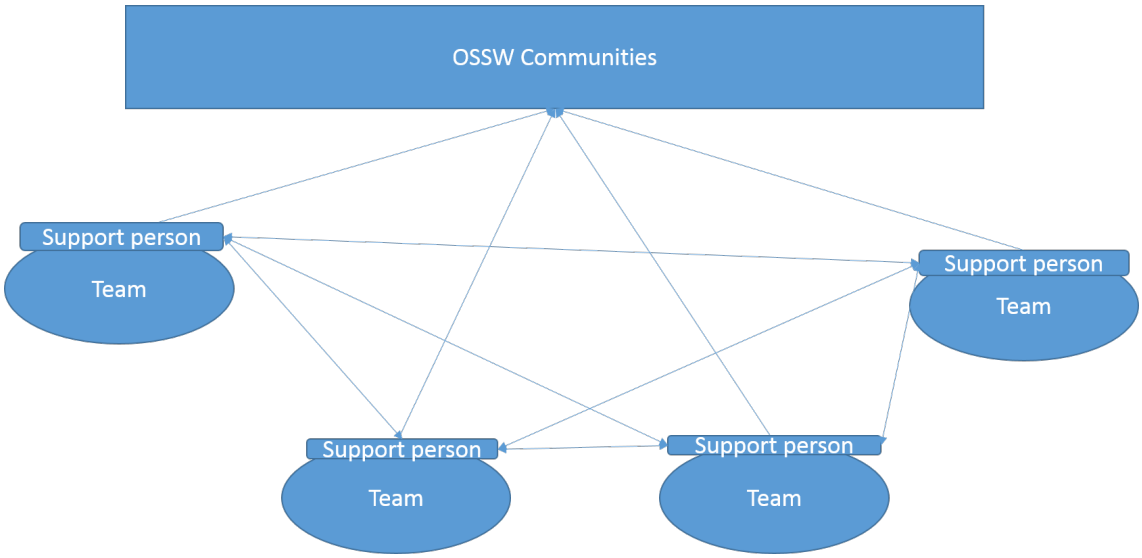

Figure 4: Distributed support method

3.6 Results of evaluating the support team methods

As already stated, we did not have time or resources to conduct any verification of any of these methods. However, we can say that most likely any of the three new methods defined in this thesis are better than the baseline method for our purposes. In the baseline method every employee becomes a semi-support person but only on top of the employees own work. In every other method, there is no stress on the employee apart from having to file a request for support to the support team.

Benefits of the baseline method include the reduced direct monetary cost of support. The support comes directly from the open source communities and costs nothing to very little in terms of money. It would most likely be much more costly to have a permanent or semi-permanent support team in place. However, nothing is free and in the case of the baseline method, the cost would be mostly time. Checking legal requirements can take considerable time depending on the support request and type. For example, a modification to some package could require heavy legal paperwork and bureaucracy, whereas a simple support request with no project critical data might get accepted much easier. In these examples, the cost would be time and the salaries of the legal staff. Also, if the open source community collapses and the project is discontinued, the baseline model cannot answer this situation. In this case, a support team would be needed to take over the project. If this were to happen, we would need to have some other method than the baseline method.

However, a permanent or semi-permanent support team would have contractual

obligation to provide support at a high quality and in a timely fashion. This cannot be guaranteed with open source communities. Another thing lacking from the communities is the insight to the purpose of using certain software together. Also, the possibility of having the support personnel seated in the same places as the products or even teams would make the support quite accessible.

In conclusion we can say that the different support methods could be utilised in different situations. The general support team method could be utilised when there would be a smaller user base and everyone would be using roughly similar Linux distribution but with small variations. When the distributions used by different products would begin to vary more, having a single large team might become cumbersome. In this case we could adopt the product support team method. Finally, the distributed support method could be integrated with the product support team method. In this integrated method, the product support team personnel could be distributed into different teams instead of having a single team. This would then bring the support to the people who need it the most, the developers.

These methods could be transitioned from and to depending on the amount of activity. If there were high amounts of division, the scenario described above could be implemented. On the other hand, in the case of reduced activity, the general support team method could again be adopted to save in costs.

A support team or teams is not enough though. We need to have an effective set of tools and processes at the support team or teams disposal. This would enable the support team or teams to perform maintenance tasks more effectively. In the following Chapters, we will go through how software is distributed in the form of software packages and what tools could be used to build and maintain new Linux distributions from those packages. The main idea is to have the support team to use these tools to fulfil a set of processes required to create or maintain a new Linux distribution from scratch.

4 Package management

Software packages have become the standard way of installing software on Linux. Packages make it easy for users to install and for package creators to distribute software. What packages and package management do, is that they hide away the heavy configuration and dependency management from the users. This makes not only the users lives easier but also is easier to the package creator as the package installation

becomes standardised. This standardisation makes it easier to, for example, help with problems as the creator knows what configurations have been done and what steps the user has taken when installing the package.

Understanding how the packages are built and managed is also important due to our corporate requirements. We need to have the packages stored in some accessible location and we need to ensure that none of the packages will create a security risk. In this Chapter we first look at how packages are managed and what packages are. More specifically, we have studied RPM packages used by many popular Linux distributions. We then take a brief look at the construction of RPM packages. Finally, we go through some points from the point of view of our corporate environment. We touch on how we could monitor the contents of the packages so no security vulnerabilities would be accidentally injected to our system. We then go through how we could arrange the storage of the packages in our system.

4.1 Package manager

Package manager is a tool for automatic management of packages. For example, package manager takes care of the dependency management of packages as well as keeps track of where the packages are in the system. While package manager can simplify the management of packages, using a package manager requires some upfront configuration work. For example, a popular package format .rpm requires creation of a .spec file for each .rpm package. These .spec files specify how the package should be installed, where the source code is located, what are the dependencies and so on. With this .spec file a user can then create a binary package and store it in some repository which can then be accessed by the package manager. In this subsection we first look into some general information about package managers and then some benefits and drawbacks of using a package manager.

Using precompiled packages like .deb and .rpm with their respective package managers can help alleviate the need of large user interaction. By using package managers the users can download the correct package directly from the package repository of their distribution. For example Fedora and Debian both host their own package repositories [FeP] [Deb]. When the new version of the package is released, the Linux distribution administrators can simply add the new version of the package to their repositories and notify the users. The users are then free to update the package from the repository using the package managers they have in their distributions. The usage of these package managers has been made very simple and intuitive so

the user effort is reduced significantly. The use of a package manager is also easy to automate as the functions of the package managers are hidden behind common commands.

Another benefit of using package managers is easier customisation of Linux distributions. The administrators would be able to create custom images for the users according to user specifications. The users could even be allowed to customise these images directly themselves. A good example of the latter is SUSEStudio which is a web page frontend to the openSUSEs Kiwi build system [SUS]. Through this users can build their own Linux distributions by selecting the packages they would want to be on the new distribution. Users can even upload their custom packages to be used in the new distribution.

4.2 Problems when using packages - dependency hells

One of the major obstacles when using any packages is the dependency management of the packages. RPM and other packages are notorious for their dependency hells. Artho et al. claim that dependencies are restrict the freedom of installation, upgrade and removal of packages [ASDC+12]. There are many different types of dependency hells, for example, circular dependencies. This means that a package depends on an another package which might, in the worst case, be directly or indirectly dependant on the original package. This kind of circular dependency can cause administrators massive headaches if not taken into consideration. Luckily, many of the modern package managers such as *dnf*, *zypper* and *apt* can detect and handle these kinds of circular dependencies.

These dependency hell defects are not the only defects and conflicts caused by packages. For example, in the article "Why Do Software Packages Conflict?" Artho et al. identify five different causes of conflict defects [ASDC+12]. They found that packages could for example conflict by having package A expose some yet-unknown problem in package B. This was found out by Artho et al. to be caused by package A and B interacting in a way not considered by the original authors of the packages [ASDC+12]. These packages could work just fine as separate entities but when together they would produce some unexpected behavior. Other problems mentioned by Artho et al include "Unavailability or Inaccessibility of Shared Resources", "Conflicts on Shared Data, Configuration Information, or the Information Flow Between Programs", "Package Evolution Issues" and "Spurious Conflicts" [ASDC+12].

4.3 Package management of Nokia Linux

In our case, the biggest problem with the current Nokia Linux is its package management, or the lack thereof. In the current Nokia Linux, there exists a rudimentary software for installing and removing other software. This package installer is not a fully fledged package manager though. Installing packages is done through tar archives but there is no dependency management.

Because the way the Hammer builds the Nokia Linux and the lack of package management tools, if a small modification is done in one of the packages, all of the packages need to be recompiled [AKH]. If Hammer would be using a package manager tool, only that package would need to be recompiled. As all the other packages and the updated package would be stored in a repository, Hammer would then pull all the ready packages and just install everything else and then doing the update on the updated package. This way, maintainers save time as they would not need to be recompiling everything all the time.

4.4 Package composition customisation for Nokia Linux

Nokia Linux package composition with the current package management gives very little room for customising Nokia Linux distributions package composition. Current Nokia Linux does not have an option for customising the composition of installed packages and if users would want to have any, users would need to maintain their own forks of the Nokia Linux. However, as there are multiple different groups of users using Nokia Linux, there are some packages that are only needed by a few of those groups. This could be resolved by having a package management system capable of customising the composition of the installed packages. However, the Nokia Linux circumvents this problem a little bit differently. The Nokia Linux contains all the packages required by all the users. For example, if one user requires Java but another does not, the latter one gets an image that contains unnecessary functionality. One package might not seem much but if there would be multiple packages unused by some users, then the images are going to get unnecessary large for these users' purposes. To answer this problem, with the help of package manager and package repositories, composition lists could be used to allow users to customise their own images without the need of maintaining an own internal fork of the Nokia Linux.

4.5 Source packages

Source packages are little bit different as these packages are offered, as the name suggests, as plain source code files. The user needs to configure, build and install these packages manually by themselves. However, this enables users to also customise these software packages more easily. Different configuration and build tools include programs like autoconf and make. Note that these tools are not package managers. Using source packages can be summed as trading convenience for greater control over the source code of a package.

When managing source packages users cannot rely on some package manager to know the install location of each file of each software package. For example, when a system has a lot of packages that are all manually installed from sources, removing one will require users to track down all of the components of the software package and remove them all. An experienced maintainer might have the locations written down somewhere but even this might not be enough. As the number of packages increases, so does the number of dependencies between those packages. When removing a package, the maintainer must also know if some file is needed by some other package. Failure to do so might compromise the other software package and possibly even make the whole system unstable. Using source packages only will require a lot of manual work especially if something needs to be removed from the system.

Another problem of using source packages arises when a Linux distribution has a wide distribution. When one of the source packages receives an update, new version or patch, that change needs to be applied to the distribution image. This new distribution image would then be distributed to users who would replace their old images with the new one. The whole process might take anywhere from several hours to even days with all the communicational delay between maintainers, distributors and finally users. Especially in larger systems, maintainers need to be sure there are no critical jobs running in a system before the distribution image can be replaced with the new one.

4.6 RPM and RPM packages

RPM is an abbreviation of Red Hat Package Manager [RHM]. However, the term RPM has become synonymous with a package installed by Red Hat Package Manager or any other package management system using the Red Hat Package Manager. In this thesis we have used the term ”RPM package” to represent a package installable

by Red Hat Package Manager or any of its derivatives. The term ”RPM” is used in this thesis to refer to any package manager, such as yum, zypper or apt, capable of installing RPM packages.

RPM packages and RPM have been studied in this thesis because this technology was chosen to be used in our proof of concept project by an earlier study conducted at Nokia.

4.6.1 SRPM packages

An SRPM package is a source package containing a .spec file and everything else required to build the SRPM package to a binary RPM package. The name SRPM package is an abbreviation of source RPM package. A build tool is used to compile a SRPM package into a RPM package. Such build tools include rpmbuild and Mock.

SRPM packages are necessary for our purposes as we want to ensure software security, independence and availability. SRPM packages enable users to do configurations and customisations in the source code before the SRPM is compiled into a RPM package. By creating their own RPM packages out of SRPM packages and then storing the new RPM packages into some repository, companies can become independent of public RPM package repositories. This is vital in order to keep the availability of the packages used by companies Linux distributions. A disruption in the delivery of new packages to a Linux distribution could cause major harm to the company or the customer. For example, during the Heartbleed incident [HaB], if the delivery of new patches to fix the Heartbleed would have been disrupted, malicious users could have been able to exploit the weakness longer. This could have compromised the security of distributions vulnerable to the Heartbleed bug. With SRPM packages available to a company, the company would be able to protect itself by implementing the bug fix to the SRPM package, build a RPM package out of it and distribute the new patch.

4.6.2 RPM specification files

RPM specification files, or .spec files for short, are used to build source code into usable RPM packages. .spec files contain all the necessary information to compile, build and install a piece of software to a system. These .spec files also specify what the package requires, or the package dependencies, and what the package provides to other packages. An example of an .spec file can be found in Appendix A. This is

the .spec file used in our proof of concept project to build the Expect package.

Creating a .spec file begins with getting the sources for the new rpm package. The next step is to use editor to create a new file with the extension .spec. For example, a common text editor, vim, creates the basic structure of a .spec file automatically, when a file has .spec file extension. The naming of this file should be consistent with the name of the package. Creating the .spec file in this manner initialises some key fields in the file and makes the overall .spec file creation process a little bit easier for the package maintainer. After this step, the actual package specific configurations can be added to the new .spec file.

In the .spec file, maintainers can for example define that the output of building the SRPM package to RPM package should be multiple rpm packages. By defining multiple packages as the output, maintainers can reduce the need of .spec files to only one per package. This way for example devel packages, a commonly used name extension for packages containing development header files and other things required for development, can be created at the same time as the regular package and with using only one .spec file.

When all the configurations have been done the new .spec file can be tested. Put the .spec file to the rpmbuild's build environment "SPEC" folder and the sources required to the "SOURCES" folder. Remember to install any dependencies to the build host. Running the "rpmbuild -bb name-of-the-spec.spec" command will begin the creation of a new RPM package. The ready RPM package can be found in the "RPMS" folder of the rpmbuild's build environment.

4.7 Our corporate environment needs - security and storage

In our corporate environment we have a need for building and storing RPM packages. We discussed these in some detail earlier in Chapter 2. Our corporate environment enforces us to build our own packages and to store them safely but in a way that is easy to access. For this we require our own build systems for building RPM packages from source code and .spec files, and for building a repository out of those packages. We have to be also able to store the source code and .spec files in a safe location.

In this subsection, we first look at how we can ensure security of the packages when we build these packages. With the integration of support personnel to the package communities we could monitor the changes made to those packages.

With storage, we propose that the source code, .spec files and the resulting RPM

packages be stored in separate repositories. This would not only give each repository a single responsibility, but also makes giving permissions to only required personnel easier.

4.7.1 Security

To ensure security and also availability, we need to be able to build the RPM packages we will be using ourselves. Building our own packages means that not only we could apply our own security patches but could apply other patches as well. However, building our own packages requires that we would have a build system for packages in place.

As the number of packages we use is large, checking each of the packages for security faults would be impractical. It would most likely take too much effort to go through every line of code to ensure maximum security. What could be done instead is to have the support personnel monitor how the packages evolve. If the support personnel are involved in the package communities, as we describe in Chapter 3, we could leverage this knowledge to ensure the source code is, at least on some general level, safe to use. Support personnel could, for example, monitor the changes made by outside contributors and intervene if something suspicious comes up.

4.7.2 Storage

We need to be able to store the packages we use in an efficient manner. In addition, we need to be able to store the source code and .spec files used for building the packages. We propose that we would use different repositories for .spec files, source code and RPM packages. This way we would divide the inputs (spec files and source code) and outputs (RPM packages) into their own clear entities. Furthermore, RPM packages should be divided still into at least two repositories: base packages and updates to those base packages. More repositories would be made when, for example, more versions of our Nokia Linux would be created. In this case, we would create a similar setup where the new version's sources, .specs and RPM packages would be in their own repositories. If there would be a large amount of the same packages used across many distributions, those packages should be added to a new common repository similar to the previous examples.

At some point, when there would be a large number of different versions of Nokia Linux in use, we would want to transition to the use of package groups. "A package

group is a collection of packages that serve a common purpose, for instance System Tools or Sound and Video. Installing a package group pulls a set of dependent packages, saving time considerably." [RHM]. These groups would ease the maintenance of the packages. For example, support teams could be assigned to focus on supporting a single group. Packages in any of these groups should be somehow similar to each other or fill a common purpose. Thus maintaining packages in a single group would most likely require similar skills across all the packages in that group.

Storing the packages in different repositories according to the Nokia Linux version would also give the possibility of limiting the access of people to the repository. For example, if we would store some of the proprietary software we are using into a repository, we might be contractually obligated to restrict the access rights from everyone else except those who really need to have access.

To summarise, we should have packages stored in repositories so that each version of Nokia Linux would have an own set of repositories. Every set of repositories should include a repository for sources, .spec files and for the RPM packages built out of the sources and .specs. These are illustrated in Figure 5. In this Figure we show a repository structure where we have a distribution with a major version with its two release versions. Both of the release versions have their own set of repositories. Keep in mind that the amount repositories for each version of Nokia Linux should be kept at minimum. This ensures consistency and also lessens the impact on the maintenance team as they then would not have to configure too many repositories into any of the distributions. The less configuration required to distributions the fewer possibilities for errors.

5 Processes for creating and maintaining a distribution for corporate environment

In this Chapter we define three use cases and then propose three processes to solve those three use cases. The first process is a process for creating new distributions and the second process is a process for maintaining these distributions and their packages. The third process is a process for creating new release using some of the packages form the old repositories. We have proposed three processes because of the subtle uniqueness of each situation. We are aware that all these processes have

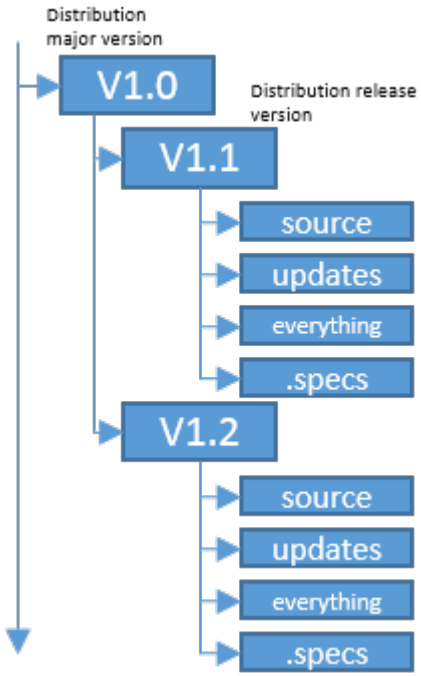

Figure 5: Example of package repository structure

a lot in common. This is why we also propose a master process, basically showing the common ideas that all of these processes have.

5.1 Use cases

The following three use cases are what we want to represent with our processes. These three use cases, creating, updating and making a new release of an image, are the core of our proof of concept project.

1. Creating a new distribution requires a new repository and new configurations to be created.

2. Updating an existing distribution requires updating the packages and then rebuilding the distribution.

3. Creating a new release of a distribution will use updated versions of some packages and some packages will be the same. We have to create a new repository for the new release.

Creating a new distribution. We want to have a clear process on how to create a new Linux distribution form a set of source files and their configurations. These

source files will be built into binary packages using their configurations. We need to store the resulting packages into a repository along with everything else required to recreate those packages. From those packages we want to create a new distribution. This distribution should also be stored somewhere.

Updating a distribution. We want to update the packages of a distribution when new versions become available. In this use case, we have the repositories for all the components ready. What needs to be done is to acquire the new source files for the package being updated and rebuild the package. This should then be stored into a repository, the image configuration modified to accommodate the new package, and the image be rebuilt.

Creating a new release of a distribution. In this case, we have most of the packages ready and most of the configurations ready for the images. However, we need to create new repositories for this case to keep the versioning of the distribution distinct. We still need to acquire some source files for the packages we might want to change or update in this new release and also update the image configurations accordingly.

Derived use cases. At this point a sharp reader might have noticed that we did not define a use case, for example, for changing the package composition. This is a conscious decision as we felt that these three use cases we presented can cover at least the other use cases we came up with. If we take the package composition modification as an example (described in subsection 4.3), we could use the release process to do this kind of modification. The new package composition would require its own package repository but would still be using a lot of the old packages. This is exactly the use case the release would be used on. We feel that with the right application of any of the three processes we could cover at least most of the special use cases. In the event that there would be some use case that we would have missed, the master process could be derived into a new process.

5.2 Requirements for executing the processes

As the practical part of this thesis is about finding a more flexible build system to replace our current ageing build system we have to define what is meant by flexibility in our case. Our definition for flexibility requires the build system to be modular and to support multiple, interchangeable technologies. This means that its components need to be able to be changed while the whole still retains the same functionalities.

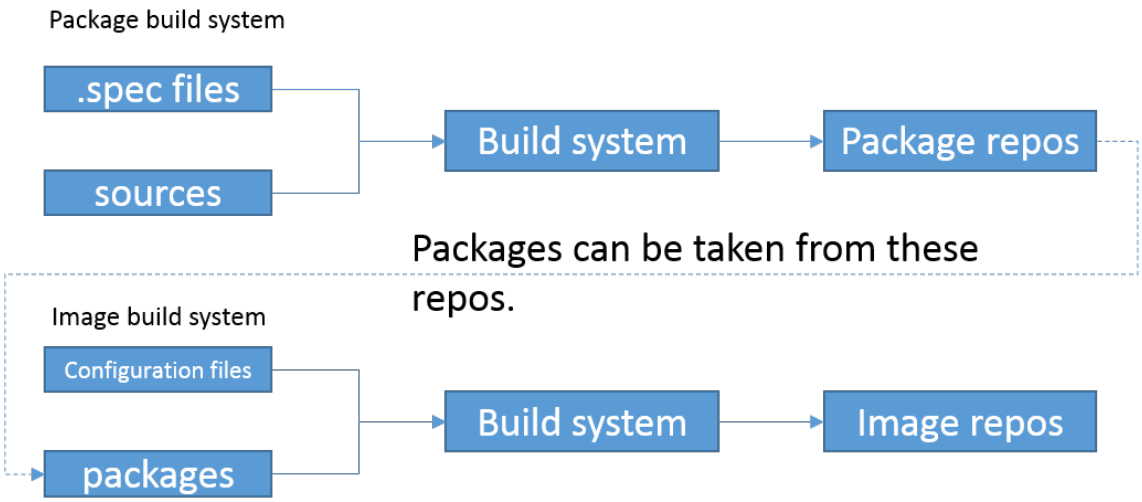

Figure 6: Distinction of build systems

Because of this we have left the processes more abstract.

The presented processes do not specify any tools. There are, however, some tool requirements. First of all, these processes require the use of binary packages. This creates two other requirements: building the binary packages require some binary package build system and those binary packages require the acquisition or creation of some configurations that the binary package build system can utilise when building the binary packages from source code.

These processes also require the creation of different kinds of repositories. The repositories for source and binary packages can be similar to, for example, RPM repositories. These repositories could be hosted on, for example, some server to make the packages readily available.

The repositories for the configurations could be similar to, for example, git or SVN repositories. The configuration files are not big in size so a git repository, for example, could be a possible place to store these configurations.

There are two different build systems used in the processes: the binary package build system and the distribution build system. These build systems do what their names suggest: the first creates binary packages from source code and binary package configuration file and the latter creates distributions from ready binary packages and distribution configuration file. The distinction between these two build systems is illustrated in Figure 6.

All of these tools and requirements can be changed to suit different company needs and limitations. For example, if a company would want to use only one specific server for all repositories, then they could do so for all the packages and configurations. These processes are designed to be abstract and requirement agnostic. The real implementation of these repositories, build systems and configurations is left to the company, team or anyone else using these processes.

5.3 Processes

To resolve the use cases presented, we first define a master process. This master process has been left intentionally abstract and has to be specialised to suit each use case. Because of this, we have refined a solution for each of the use cases from the master process. Thus, we have three processes: Process for creating a new distribution, process for updating a distribution and process for creating a new release of a distribution. Each of the refined processes is a complete set of actions required for fulfilling the use case.

These kind of processes are not new though. For example, Fedora has been using a similar process for a long time, at least seven years. The implementation of that process is the Koji tool, first created in 2010. For our purposes we would like to have something similar. The Fedora model is quite appealing as it has been in use for a long time and has been tried and tested.

5.3.1 Master process for creating and maintaining a distribution

This process illustrates the general overview on how each of the following, refined processes work. This ”process” is more to show the very high level ideas that each of the processes have. All the other processes have been refined from this process. This process is not used in any of the proof of concept tests. This process can also be thought of as the framework process. We present this process to show our thought pattern and to record how we came up with the processes we used in our proof of concept project. Three use cases have been refined from this process.

1. Create a set of packages. In this step we create a set of binary packages. This step requires source code and configuration files and outputs the packages.

2. Store the previous steps inputs and outputs. In this step we place the created packages and the inputs into repositories. This step takes as input the outputs

and inputs of the previous step and store them into a repository.

3. Use packages and build system configuration to create or recreate a distribution. We use the repositories created in the previous step and some build system configurations to output a Linux distribution image. This step takes as input the build system configuration. In these configurations the used packages and possibly the repositories of these packages are specified. This step outputs a new distribution image.

5.3.2 Process for creating new operating system distributions and their images

This is a process for creating a brand new distribution from source packages and configurations. This process should be relatively rare as it would be required only if there are no existing distribution repositories. This creation process requires the source code for all the packages about to be included in the new distribution. In addition, some location is required for the different repositories. The configurations for the packages and the image need to be created or acquired.

1. Acquire source code and acquire or create package configurations for all packages.
2. Using the source code and configurations, create binary packages.
3. Store the newly made binary packages into a repository, the source code into another repository and the configurations into a third repository.
4. Create a new configuration for a build system.
5. Store this build system configuration into a repository.
6. Use a build system to build a new distribution from the stored binary packages and the build system configuration.

5.3.3 Process for maintaining existing packages of a distribution

In this process, we are updating some packages of a distribution. If most of the packages of a distribution would be upgraded, then a new release should be made instead of a simple update operation. This update process requires the sources of the

packages being updated. Also, no additional repositories should be needed as the inputs (sources and configurations) and outputs (binary packages and distribution images) should already have some repository where to update the respective items into. This is due to

1. For all the packages involved in the maintenance, modify configurations if necessary and apply updated sources.
2. Using the updated sources and configurations, recreate updated binary packages.
3. Add the new versions of the binary packages to the same repository as the old ones keeping the old versions. Do the same with sources and configurations with their respective repositories.
4. Modify the build system configuration, if necessary.
5. Store the updated build system configuration in a repository.
6. Using the updated binary repository and build system configuration recreate the distribution.

5.3.4 Process for creating a new release of a distribution

This process includes steps similar to the two earlier processes. This process would be used when there are enough updates to many packages that the distribution would be considered a new distribution. At this point a new release should be made instead of just continuously updating the packages. However, this new release is still likely to contain packages found in a earlier release of this distribution meaning that we wouldn't be starting totally from nothing.

1. Create new repositories for binary and source packages, package configurations and build system configurations. These repositories should be named according to the new release version.
2. Fill these new repositories with desired packages and configurations accordingly.
3. If necessary, recreate binary packages using sources and package configurations and add the recreated binary packages to their correct repository

4. Modify the build system configuration, if necessary.

5. Store the updated build system configuration in build system configuration repository.

6. Using this new binary repository and new build system configuration, create a new distribution using the build system of choice.

5.4 Problem of architectural change

When designing these processes we had to keep in mind architectural change in our own production code as well as in the build system itself. All software changes and evolves over time unless it has become obsolete. In order to create a build system for our purposes we had to take into account how our software might change overtime. For example we might want to rely on a package that has been installed in a specific way. As we are trying to create a general process and a modular build system we want to design these so that they are abstract enough to accommodate for architectural changes at least the small ones.

In our proof of concept project we have taken architectural change into account by matching the existing Nokia Linux distributions development packages and their contents. This was also our greatest challenge due to current Nokia Linux using source packages and the build system using RPM packages. The main issue was that not all of those source packages were available as RPM packages, had different names or had only a differing version available.

Our processes are already abstract enough to accommodate for architectural change. The only real limitation that the process imposes is the requirement of having binary packages. Otherwise the format of the packages does not matter, the storage method is not defined and the Linux build system can be changed. In our proof of concept project we used different tools for building the packages to provide an example of how the different tools can be changed. The entire process could also be accomplished with a single complete build system like Yocto or Koji. As long as users of the processes presented in this thesis are using binary packages stored in some repository from which a build system uses them, then there should be no problem with architectural change. However, this is still at this point, pure speculation. We do not have the data to fully claim that the architectural change in the production code would not affect these processes in a major way. To solve this uncertainty, we

propose a longer term study on how architectural change would actually affect the processes presented in this thesis.

6 Tools used in our proof of concept

A build system is a tool of automation. It automates a building of a software from a begin state to a target state defined by the user of a build system. This automation makes the software easily reproducible as the building process becomes standardised. A build system also eases maintenance of software. Users can experiment with the code of a piece of software and then just use the build system to build that code into the software for final testing. Adams et al. (2008) define two key purposes of build systems. They state that "A build system takes care of two things [ADSTDM08]:

- it decides which components should be built and establishes any platform-dependent information needed to do so;
- it incrementally builds the system taking dependencies into account."

The first build system was introduced in the 1977 when Feldman created the make build tool. Before this people were relying on their own build and install scripts [ADSTDM08]. The "make" tool was notable due to its use of "makefiles" which defined the dependencies of the targets and then ran a set of shell scripts on them to build a target. However, "make" was not a complete build system. According to Adams et al. portability has always been of concern with build systems as for example compiler versions might differ between computers [ATDSDM07]. To solve this problem, configuration systems, like the GNU Build System, were created. The purpose of these systems is to make the configurations more abstract making the use of build tools less platform-specific and more generic [ADSTDM08].

Build tools and systems come in varying shapes and sizes. Examples are make, Ant, Gradle, Koji, Kiwi and Hammer. Out of these examples, the first three are build tools and systems for mainly building software out of source code. The latter three are build systems for building entire Linux operating system distributions. In this thesis we are more interested in the Linux build systems as well as build systems for building binary packages out of source code like the rpmbuild and Mock build systems.

In this Chapter we first take a brief look at Hammer and the two Linux distribution

build systems we were using in the proof of concept project, Koji and Kiwi. We then explore two RPM package build tools, Mock and rpmbuild, and two RPM repository creation tools, Mash and createrepo.

6.1 Hammer

Hammer is the current build system for the Nokia Linux. It is a LFS (Linux from Scratch) like build system using source packages either from a git repository or from tar-balls to build a monolithic, custom Linux [AKH]. With this kind of package delivery system Hammer is able to maintain the latest versions of packages custom built into a working Linux distribution.

Hammer has been released as an open source project and can be found on git [AKH]. The build system was designed to be as easy to learn and intuitive to use. Hammer also supports its own types of packages, but these packages are more like large modules. Examples of Hammer packages are ”rootfs” and ”toolchain” containing the entire Linux root file system and Linux toolchains respectively.

6.1.1 Advantages of Hammer

The main advantage of using Hammer is its simplicity. This is also a part of its downfall. The main advantages of this simplicity are the ease of use and configuration. You only require the locations of the source code of the software you are about to use. This location can be, for example, a git repository. Hammer then uses user defined scripts to compile and install the packages. This means that Hammer does not need to do the extra work of creating binary packages and then storing them into some repository. This also means that Hammer can always use the latest versions of the source code. With binary packages, users would either need to wait for the official configuration files to be able to build the binary packages, or create their own configurations. However, with Hammer you still need to have the custom scripts in place somewhat diminishing the benefits of this point.

All in all, Hammer is simple, easy to use and gets the job done. However, we would want Hammer to do more. Instead of trying to reinvent features already present in other Linux build systems and incorporating them into Hammer, we decided to first have a look at what could be done using other Linux distribution build systems.

6.1.2 Limitations of Hammer

Currently, the greatest limitation of the Hammer build system is the lack of support for proper package management. As explained earlier, Hammer does use its own version of packages but it does not have a package management system like RPM or dpkg. Now, as there is no package manager installed by Hammer, users cannot easily install, remove or manage packages in their systems. To install packages, users can use, for example, makefiles.

To manage already installed packages, a package manager would have to already exist in the system before installing other packages. Otherwise, as the already installed packages are installed directly from sources, no dependency info would be present, for example. Usually, Package managers need this dependency info to build a dependency graph. Currently, with most package managers, this graph is used to track which package is depending on which package. Hammer does not have a package manager utilising any automatic dependency management, meaning that one of the best reasons to use a package manager, is lost in Hammer built distributions. The package manager would then be a tool able to just install and remove packages by executing some configuration file, a task already done by, for example, using makefiles. Hammer would need to be modified to be able to install all the packages of a distribution using some package manager during the distribution build process. If this were to be done, the package manager would then have the dependency graph and all the installed packages would be under the package managers administration. Still, as stated before, we do not want to reinvent the wheel. Suitable Build systems like this already exist and we should first study these systems before attempting to modify Hammer.

The reason to have a package manager is that the package manager gives the option of easy customisation. Otherwise, we couldn't easily install packages for different user needs. Packages wouldn't need to be compiled at the same time for each distribution, but could have a repository containing the most common packages. Having a repository for packages enables the package manager to install packages directly from this repository without having to compile everything locally. As these packages are most likely built on different platforms, we also need cross compilation.

Meeting this requirement, however, is not possible with Hammer, as Hammer does not support cross compilation. Making a cross compilation is an action where a piece of software is compiled on some host machine to be compatible with some target machine different from the host machine. Effectively, this means that anyone

can build packages for everyone. Respective to cross compilation, making a native compilation is an action where some host machine compiles a piece of software to itself. This natively compiled package would be only usable in nearly identical distributions as the host machine. It is possible to modify Hammer to support cross compilation but again, we are going to first look at already existing solutions.

Moreover, the lack of cross compilation creates the following problem. Every Hammer built distribution must be built on an earlier Hammer built distribution, as Hammer compiles its software using the native compilation. In addition, Hammer expects users to do this on qemu virtualisation environment. If the host machine running the qemu virtualisation environment does not have resources to spare, the build can take hours to finish.

Another limitation is the result of a build when using Hammer. Hammer builds monolithic distributions. Earlier in this thesis we already explained why this is not desired for our distributions. To recap, all the packages in monolithic distributions need to be rebuilt every time a modification is made to one of the packages. This causes unnecessary work and delay for distribution maintainers, users and the users' customers.

A solution for these limitations would be to modify Hammer to incorporate a package manager. The package manager would need to be installed with the help of the host systems tools. After the package manager would be set up this package manager could then be used to install every other package. This would enable the package manager to be able to manage all the packages. Also, by using a package manager the created distributions would be modular instead of monolithic.

A great limitation of Hammer is, despite it promises to be an easy to learn tool, it seems that there is no documentation other than the code itself. This can make newcomers struggle as there is no real examples of how to use Hammer.

Hammer has a lot of limitations but all of those could be fixed. However, as we stated, we have focused on studying alternatives instead of fixes to Hammer. As there are many alternatives to using Hammer, why reinvent the wheel?

6.2 Koji

Koji is the build system used by the Fedora project to build all the RPM packages and distributions for the Fedora Project [KoP]. It is a powerful build system tool designed to provide its user with all the necessary tools to create a new Linux

distribution from source packages. For this it uses an architecture of different servers of hubs and builders called Koji daemons. Despite its powerful nature, it is quite hard to get into due to requiring a lot of different competencies from the maintainer. To alleviate this the Fedora project hosts a public Koji build server. However, we require our own build system that would be maintained by ourselves internally to ensure security and control the quality of not only the outputs but inputs as well.

There are five main components in Koji. Koji client is a command line interface to the Koji hub. Using Koji client, users can for example start new builds with Koji. Koji hub is a manager that manages the tasks given by users. When a user creates a new task through the Koji client, the Koji hub takes that task to a database and tags it as a free task. Kojids are worker daemons that periodically poll the Koji hub for free tasks. There can be many Kojids linked to a single hub and these Kojids can be ran on different computers. This makes Koji quite scalable. These three elements are used to execute the tasks assigned to a Koji build system [KoP].

Other than Koji client, hub and worker daemons, Koji also has a component for showing the build results on a web page, Koji web, and a component to keep the build root repodata updated [KoP], Kojira. Other than these five components, important parts of Koji also include a database to keep track of the tasks and a shared storage for the build results. The relations of these five elements can be seen in Figure 7. It is a simplified figure of the build process.

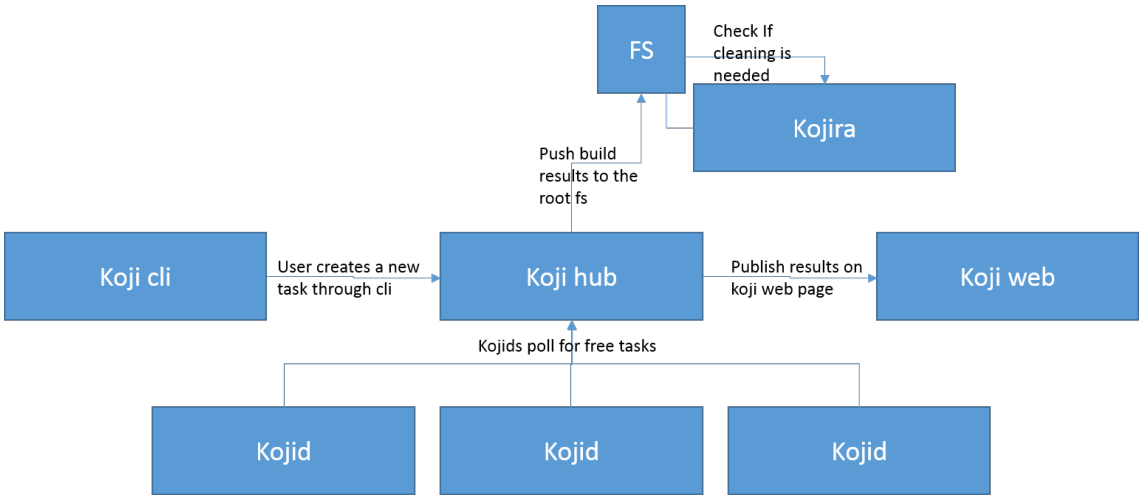

Figure 7: Simplified Koji build process

Koji works fast being capable of building new, minimal Linux images in just about 5-

15 minutes depending on the configuration. The configurations are given as Red Hat kickstart files [RHK]. These kickstart files define the package composition as well as some general system configurations like language, timezone and firewall settings. At the end of a kickstart file there is a space for some scripts that are ran at different points of the build.

For building RPM packages, Koji uses Mock. Mock is a tool that builds binary packages from source packages in a chrooted environment. These packages are then stored in the Koji's internal cache. If a user would want to publish these packages, a tool called Mash can be used to automatically create a repository from packages created with Koji. User needs to use Koji to first tag the desired packages and then Mash will find those packages in the Koji cache and then build a new RPM repository complete with source repository as well.

The steep learning curve of Koji comes from the requirements for the maintainer. In order to setup a private Koji build server, a maintainer would be required to at least know how to create and maintain SSL certificates, postgresql databases, Apache configurations and have a basic understanding on how yum, mock, Koji and mash tools work together, just to name a few [KSe]. Maintainers would also have to setup all the many different components of Koji. Especially setting up the Kojid scalability is going to take more effort as Koji itself offers no scalability tools for this kind of activity. Koji project does provide the maintainer with a documentation on how the system is then setup. Some scripts made by third parties can be found in github but these seem a little bit outdated.

Koji is going to store a large amount of information from each build including build logs, binaries and environment information. Storing basically everything from every build is going to bloat the shared storage quite fast to a large size. This is something that Koji developers themselves warn users about [KoP].

Documentation for Koji seems to be somewhat scattered. There is some documentation on the Fedora wiki page and more on the Koji projects own pagure page. Also there were a couple of typos in the commands provided by the documentation in the time of writing this thesis.

6.3 Kiwi

Kiwi Image System is an open source tool for creating Linux images in a variety of formats [KIS]. To create images, Kiwi uses a two- step process. First, an unpacked

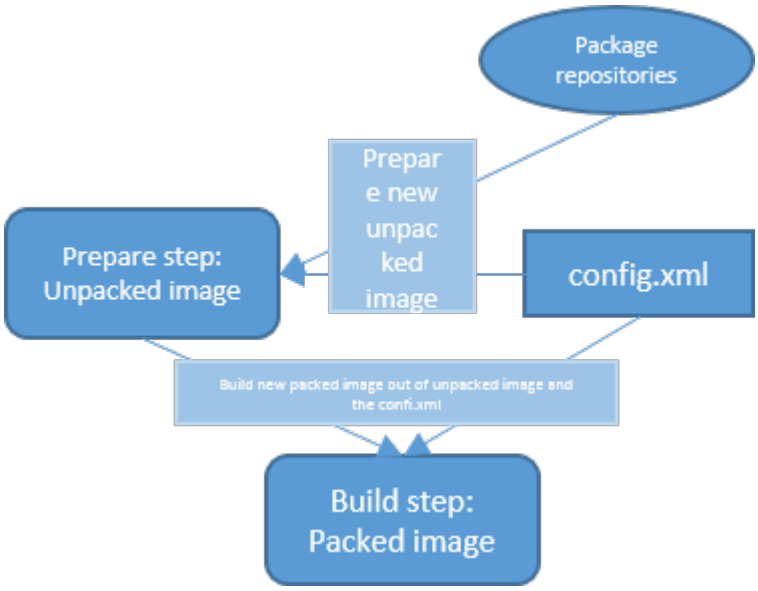

Figure 8: Kiwi build process

image is created using a Kiwi boot configuration file. This boot configuration file lists the needed packages and their respective repositories. Using this unpacked image and system configuration file, a packed image is created in the specified format. This packaged image is then provided to the user [KIS] [CLK]. The reason why the build process is two-phased is to enable the creation of multiple distributions with different formats in a single run of Kiwi. These distribution formats include qcow2, iso and docker container image formats. This speeds up the creation of multiple distributions as the prepare step is only ran once for all the distributions and not once per distribution. This two step process is illustrated in Figure 8.

6.3.1 SUSEStudio - Kiwi frontend

Kiwi is a purely command-line tool in itself but has a web frontend in the form of SUSEStudio [SUS]. This tool can be used to build images with Kiwi or just to create configurations that are then used by some local machine running Kiwi. However, SUSEStudio does provide only limited customisability compared to only using Kiwi.

SUSEStudio could make starting out with Kiwi easier. Users can use the SUSEStudio to create images and configurations without having to completely start from scratch. In other words, SUSEStudio can be used as a ”playground” to test out the capabilities of Kiwi. For our purposes we could use SUSEStudio in this ”playground”

manner as a learning platform on how Kiwi configurations are made and what they include.

6.3.2 Limitations

Language requirements of Kiwi include xml, shell and Kiwi's own functions. This can be mitigated by the use of the web frontend SUSEStudio. Kiwi's own functions is a set of common functions that can be utilised in the creation of config.sh and image.sh files. These files are used, for example, to do automatic network configurations for the distributions.

Kiwi itself runs best on openSUSE Linux distributions. Kiwi could be installed to Fedora, for example, but this would require Kiwi packages installed manually. If Kiwi would be needed to be automatically installed, then the packages of Kiwi would need to be modified to use the fedora versions of their dependencies.

Kiwi uses ready binary RPM packages to build the distributions from [KIS]. These RPM packages need to be in a RPM package repository to be reachable by Kiwi. This creates two limitations. Users need to create RPM packages if they want some custom functionality to their distributions and users need to create a repository for these custom RPM packages to be able to use the created RPM packages.

6.4 RPM package build tools - rpmbuild and Mock

As Kiwi is only a distribution image build system and does not incorporate any package build systems, we need another tool. Also, Koji is somewhat similar to Kiwi as Koji also relies on an outside tool to build its packages. Neither can do exactly the same as hammer. Unlike Kiwi, Koji does provide the user with an option for building RPM packages: Mash. In this subsection we have also taken a peek at rpmbuild, which we also utilised in our proof of concept project.

Mock is a tool for building RPMs out of SRPMs [MCK]. Mock will take the SRPMs and build them in a chrooted environment. chroot changes the root directory of the calling process to the location specified by the user [CHR]. Mock uses chroot to create a new clean root directory and builds packages on top of that [MCK]. This ensures that the packages are always built in a clean system. Mock uses preconfigured system configuration files to enable users to build packages for many different systems. This means that all the packages can be natively compiled. The

immediate disadvantage is that when a new system would require packages to be built, a new system configuration file would have to be made. Luckily Mock already supports a wide variety of current systems.

rpmbuild [RBM] is a tool for building source and binary packages whereas Mock can only build binary packages. In fact, rpmbuild is often used to build the SRPMs used by Mock. From the rpmbuild man pages: "rpmbuild is used to build both binary and source software packages. A package consists of an archive of files and meta-data used to install and erase the archive files. The meta-data includes helper scripts, file attributes, and descriptive information about the package. Packages come in two varieties: binary packages, used to encapsulate software to be installed, and source packages, containing the source code and recipe necessary to produce binary packages." [RBM].

6.5 RPM package repository creation tools - createrepo and Mash

As Kiwi does not provide any RPM creation or storage options and Koji uses its own internal package cache we needed some tools to create the repositories for the RPM packages. In the end we chose createrepo and Mash as they were easy to use and provided the services we required.

createrepo is a tool for creating metadata RPM repositories [CRM]. It takes as input a folder with a set of RPM packages and creates a repository out of those. This repository could then be hosted on some server for easy access for users or even kept on the computer for local use only, if such a need would arise. It is simple to use, requiring only a path to the RPMs as an input.

Mash is a tool for automatically creating RPM metadata repositories out of a Koji cache. From Mash wiki: "mash is a tool that creates repositories from Koji tags, and solves them for multilib dependencies." [MSH]. With this tool we could create RPM repositories that could then be used by Kiwi as well.

6.6 Other Linux build systems

We also tested or tried to test the following build systems. In the end though, we decided to go with Kiwi and later with Koji due to the ease of use and compatibility with our existing systems. However, we do believe that these systems could be also

utilised with our processes. Yocto is similar to Koji in the sense that it is a complete build system offering tools to build source code into a new distribution. DiskImage-builder is similar to Kiwi and would require a RPM package build tool as well as repository creation tool.

6.6.1 Yocto Project

Yocto Project is an open source project created and maintained by the Linux Foundation [YOC]. The Yocto Project aims to create a suite of templates and tools to enable the creation of custom Linux distributions for different systems. Yocto provides the tools to build Linux distributions to many different architectures, such as MIPS, ARM, x86 and PowerPC. Yocto provides a full suite of tools to build a new distribution from source packages. Yocto takes as input source packages and using those source packages builds the new distribution from ground up. Yocto in itself is nothing but a collection of tools, it is not the tool itself.

There are several components to Yocto. OpenEmbedded is the build system and is responsible of building the distribution. BitBake is the tool OpenEmbedded uses to create packages. Toaster is the web interface of the BitBake. Finally Poky is the reference distribution of Yocto project. It is more of an example project that will help new Yocto users to get into Yocto.

OpenEmbedded is the distribution build system. Like with Kiwi and Koji, OpenEmbedded takes in packages that the user has listed in a configuration file. However, these packages are provided by Yocto itself from Yoctos local cache of packages. When BitBake builds packages it places them in this cache. OpenEmbedded then retrieves the packages it needs form this cache. This makes the build considerably faster compared to a system like Hammer which builds distributions from sources every time without any caching. This OpenEmbedded build process is shown here in Figure 9.

On the left side, configurations are fed into the build system. From the top, source code repositories are being fed to build system. These are then built into packages that are stored into the cache. Using these packages and the build system, a new distribution image can be created.

BitBake is the package build system for Yocto. It takes as input a source package and a ready made Yocto recipe. This recipe acts like a .spec file in the sense that

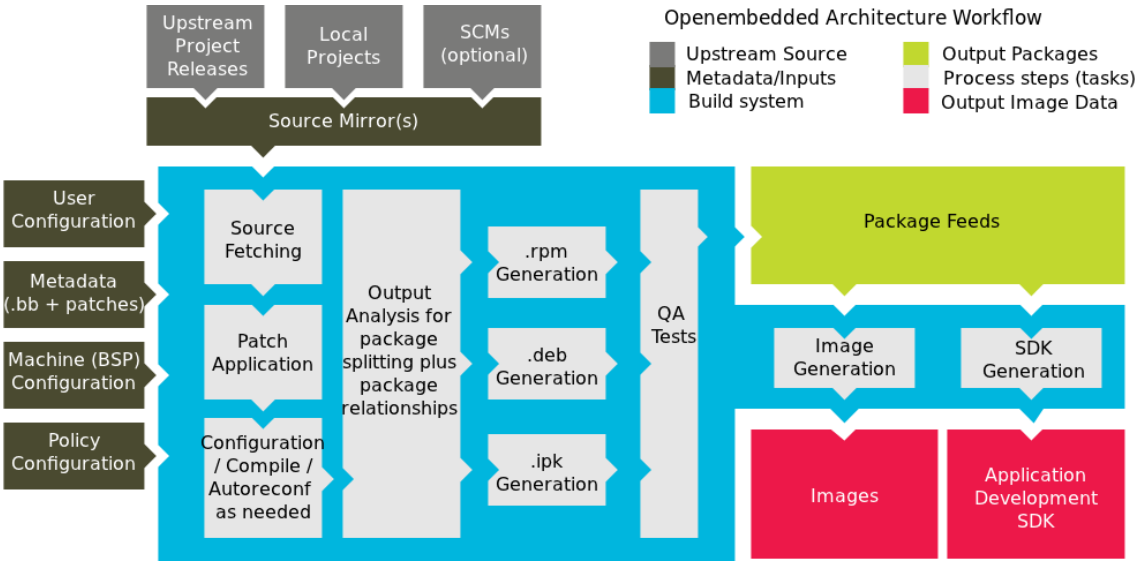

Figure 9: Yocto's OpenEmbedded build process [YQG]

it guides the building process. BitBake then outputs the package in desired format, for example as RPM.

Yocto has been criticised to possess a steep learning curve. For example in the article "Deciding between Buildroot & Yocto" Alexandre Belloni, a Yocto developer and trainer at Free Electrons told that the Yocto project requires the knowledge of at least Python, shell scripting and Yoctos BitBake specific custom language just to use the recipes of Yocto [YVB].

The recipes of Yocto are similar to .spec files of the RPM packages. We have listed this as an limitation in this thesis as creating these recipes for all the packages would most likely require more work than just reusing the ready RPM .spec files. Not only that but we would also have to learn the recipe language first. We might be able to acquire some recipes from common repositories but still there will be RPM .spec files to most of the packages we would need.

6.6.2 Diskimage-builder

Diskimage-builder was created by Open Stack to create images for their stacks [DIB]. This tool can be used to create images of different types including qcow2. It is a lightweight distribution image build system that can for example create a new fedora image out of the box.

Diskimage-builder uses its own elements to build new images. These elements con-

tain the necessary information to create a new image and can depend on other elements. These elements are a set of scripts that run at different parts of the chrooted Disk Image-build build process.

Diskimage-builder is limited to use its own elements that contain one or more pieces of software. This makes the configurations more rigid than when using a directly package based configurations as there needs to at least be a basic folder structure pointing to the correct point in the build sequence for each element. This basic folder structure requires at least two folders and a configuration file as well as a readme file according to the best practices guide of OpenStack [DIB].

7 Proof of Concept project test setups

For this thesis project, we created four different tests to test the processes we defined in Chapter 5. The first three tests were testing the creation process and the fourth was testing the update process. We did not do testing on the third process, the new release process, due to time and resource constraints. Also, as the third process is a mix of the first two, we felt that we were already testing large parts of the third process. The third process required the creation of a completely new image, just like in the first process. on the other hand, the third process can recycle some of the older packages, acting like the second process.

For these tests we used various open source tools. For our package creation tools, we chose Mock provided by our Koji server and rpmbuild to test package building locally. Our repositories are created using the Mash tool provided by our Koji server as well as using the createrepo tool. Our distribution image build system was openSUSEs Kiwi.

7.1 Kiwi configuration

Kiwi is configured using a file named config.xml. This file contains all the necessary configurations including for example packages to be installed to the new system, chosen package manager and initrd. In this subsection we have studied different important points when configuring Kiwi. We have studied what high level elements are included in the config.xml file and some specific options and fine tuning possibilities.

The config.xml divides the configurations into nine different xml elements: description, preferences, profiles, instsource, users, drivers, strip, repository and packages.

These are children of the root element, image. Out of the nine configuration elements, description, preferences, repository and packages are required to create a usable config.xml. The config.xml used in our proof of concept project sans users and packages can be seen in Appendix B. Also, the Kiwi script configuration file, config.sh, is presented in Appendix C.

Description element contains necessary information about the creator of the config file. The name of the creator, a contact method (email for example) and a short specification on what the config.xml is for, are given. As an attribute to the description element, the type of the config.xml is defined. With the config.xml file, users can create two types of distributions: normal system images and boot images. System images are what users will be using, in other words, system images are the distributions. Boot images contain the initrd that is used in system images. Kiwi provides a few pre-made boot images for the user but the users can of course create their own boot images. The creator of the config.xml thus specifies the type attribute to be either ”system” or ”boot” depending which type of image the user wants to create. We used a ready made rhel-07.0 boot image and created our own custom system image.

Preferences element specifies some of the general options for the to-be image, like what package manager will be used. In preferences element, users can also define the type of their image more specifically. We define the type to be qcow2 and the package manager to be Yum package manager. In this element, some RPM options are set to be used by the package manager. We define the following: Set rpm -check-signature and rpm-force to false. We want to use our own custom repository which has our own custom packages. These packages have not been signed so we need to tell the package manager that we are using unsigned packages. Additionally, we set the rpm-excludedocs to be true. Docs in our case bloat the distribution image unnecessarily so we want to exclude them.

Repository elements specify the package repositories that are configured to and used by the package manager specified in preferences element. We have specified our own repositories to be used in this element. This demonstrates the capability of using corporate rpm repositories with Kiwi.

The packages element contains all the packages that are to be installed to the new distribution image. In practice, there are at least two packages elements defined in the config.xml: one for normal packages and one for bootstrap packages. We have used two packages elements in our proof of concept project config.xml, one for our

regular packages and one for the bootstrapped packages.

From the optional elements we have used the users element. This element specifies the default user accounts of the distribution image. In this image we have specified the user accounts required by our environments by default. For all users we also specified their passwords in this user tag. In the user tag, a maintainer can specify user's password either in plain text or in encrypted format. Kiwi also provides a password encryption tool to encrypt user passwords. This is done using the "kiwi –createpassword" command.

7.2 Koji configuration

Koji uses Red-Hat kickstart files to build new Linux distribution images. These kickstart files are written using kickstart specific commands and bash scripts. These files can be divided into three sections. In the first part, image specific configurations are made. These include setting users, keyboard layouts and used package repositories. Our kickstart file without packages can be seen in Appendix D.

The second section is a list of packages. All kickstart files must include the @Core package, which contains some of the key components of Linux. This is included by default if the user does not include it. Users can exclude this group using a special -nocore option but this should only be used for containers [RHK]. @Base group is another default package group. Excluding this group should be done when a minimal installation is required. In this package section we included our required packages and modified the list based on what was available in our internal clones of the Fedora 25 repositories.

In the third section, custom bash scripts are defined. These scripts can be ran as pre or post installation scripts. This section is not strictly required. In this section we perform, for example, network configurations.

7.3 Testing the new images

For testing we used the same automated testing environment used by the current Nokia Linux. This ensures that the new distribution image we create is as capable as the current Nokia Linux. This automated testing environment consists of multiple automated test jobs, each installing some components of our product. Basically we are testing to see if our product could run on top of the images we have created.

7.4 Test scenarios

Just building a distribution image is not enough though. We want to make sure that an image can be built using our own repositories, our own, custom packages, with similar quality and with using our current production testing environments. For this we needed to create Kiwi and Koji configurations that would not only create a new distribution image using our own packages and repositories but also can pass through our automated testing environment. For this proof of concept project we created five test scenarios. All of these have the following objective: Test if the appropriate process can be used as intended. For example, we will test that creation process can be used to create a new Linux distribution with the desired packages from our repositories. Also, a common secondary objective for the first four tests is to test if the created distribution image can be used directly in our production environment. For the fifth test, the secondary objective is to test if the updated package works as intended. This was tested with one of our product's components. A custom .spec file was created for this purpose.

7.4.1 Testing the creation process using ready openSUSE repositories and Kiwi

The first test of our proof of concept project was designed to just creating the configuration. As the packages in the distribution are all open source, we used ready RPM package repositories to test our configurations. Otherwise, creating or finding and adapting .spec files for all the required packages would have taken too much time. The outline of our testing is illustrated in Figure 10. We have a Kiwi configuration stored in our git server that is then served to the Kiwi. Kiwi will use this configuration to construct a new distribution. The packages will be pulled from common repositories which is represented by the "Rpm repo" cloud. This new distribution image is then sent to the automated testing to be verified. After passing this verification, the image would be ready to be used in production.

The creation process was adapted to use a ready repository for this test. As we were using the openSUSE repositories in this test, we did not have to create our own packages. This allowed us to focus on creating a working Kiwi configuration. We specified the tools as follows:

1. Ready packages are acquired from the upstream openSUSE repositories.

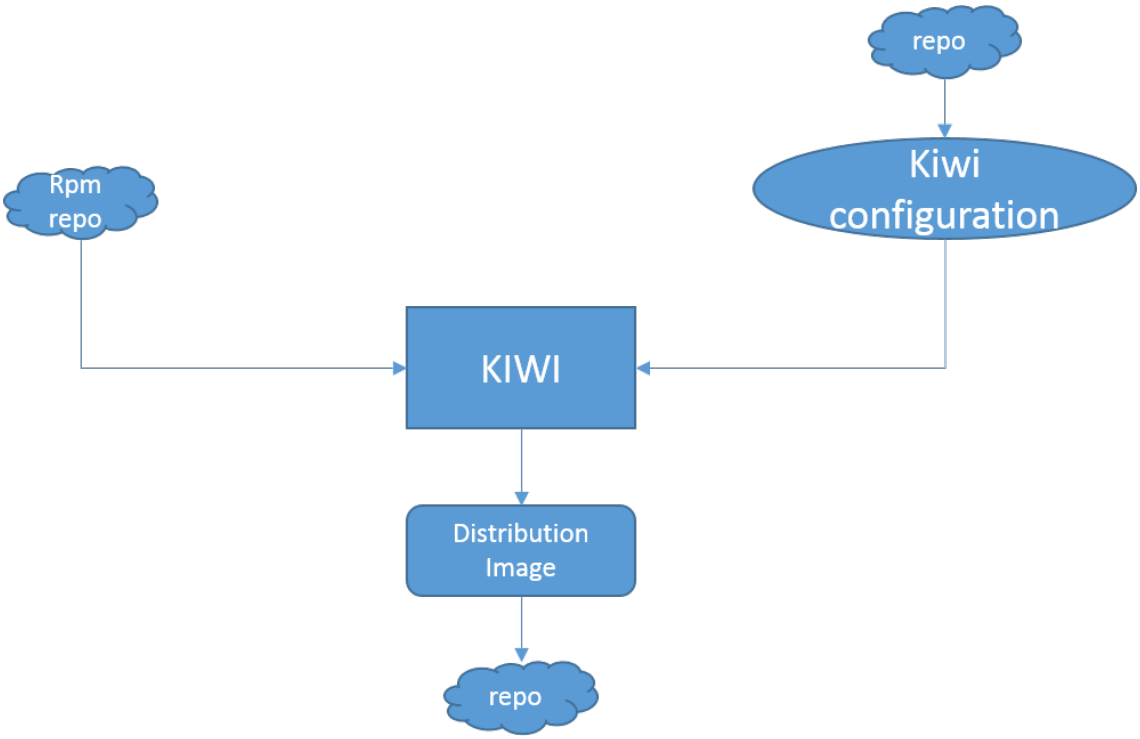

Figure 10: PoC for Process 1, test 1

2. Packages are already built so there is no need to build them again.
3. Repository is ready as well.
4. Create a new Kiwi configurations for Kiwi.
5. Store Kiwi configurations to a git repository.
6. Use Kiwi to build a new distribution from the stored RPM packages and the Kiwi configurations.

7.4.2 Testing the creation process using createrepo, rpmbuild and Kiwi

After we had the Kiwi configuration in place, we replaced the common RPM package repositories with two computer-local repositories. This means that the repositories were locally available to the computer that was running Kiwi. These repositories were created with a script presented in Appendix E. The outline of this test has been illustrated in Figure 11. Most of the RPM packages were taken from outside repositories but some were compiled using the rpmbuild tool. For example, the expect-package was compiled from sources using a .spec file and added to the

repository. The .spec file used can be seen in Appendix A. We then used the Kiwi configuration we had created earlier to build a new distribution image and sent it to the automated testing to be verified.

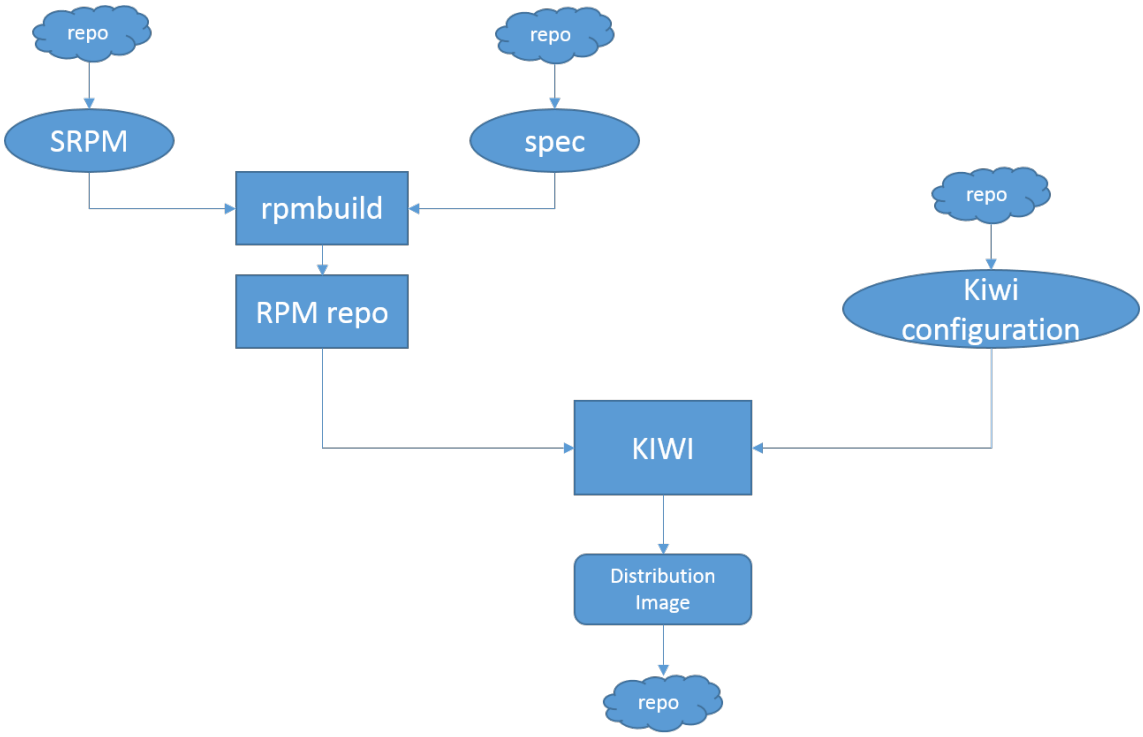

Figure 11: PoC for Process 1, test 2

Compared to the process of the first test, now we have built some packages using rpmbuild and stored them in an internal repo created by createrepo. The steps of this test were as follows:

1. Acquire sources and acquire or create .spec files for some packages. Most of the packages are taken from internal repositories.
2. Using the sources and .spec files, create RPM packages using rpmbuild. Most of the packages are still already built.
3. Store the newly made RPM packages, the sources and .spec files to a local RPM package repository. Repositories are created using createrepo.
4. Create a new Kiwi configurations for Kiwi. Add the local repositories.
5. Store Kiwi configurations to a git repository.

6. Use Kiwi to build a new distribution from the stored RPM packages and the Kiwi configurations.

7.4.3 Testing the creation process using Mock, Mash and Kiwi

The third test was done using a local RPM package repository that was not visible to any other entity. This was done to see what modifications had to be done to the build chain if our own RPM package repositories would be used. In the third test we took the RPM package build tools and the package repositories to a different computer. At this point we had managed to get a working Koji installation and also wanted to test that as a package build system. Figure 12 illustrates this. We used Mock tool used by Koji to build packages out of sources (SRPMs) and .spec files. After this we used Mash tool to automatically create a new RPM package repository from the Koji's internal package cache. Our Mash configuration file, required for the mash to know the correct cahce to pull the packagaes out from, is presented in Appendix F. This new repository was then hosted using Apache to be accessible to Kiwi. We then used the same Kiwi configuration as before and added the new RPM package repository to it. Note that we did not compile all the packages that were required for the Kiwi build. Most were still in the local RPM package repository. We did have several packages in this new RPM package repository and also some proprietary packages.

We are still using mostly the same local repository as used in the second test. These are not listed in the process as they are seen as redundant and unnecessarily cluttering the process. New packages are created and stored using Koji server's Mock and Mash. These are the steps of this test:

1. Acquire sources and acquire or create .spec files for some packages.
2. Using the sources and .spec files, create RPM packages using Mock.
3. Store the newly made RPM packages, the sources and .spec files to a our own, public RPM package repository. Repositories are created using Mash.
4. Create a new Kiwi configurations for Kiwi.
5. Store Kiwi configurations to a git repository. Add the Mash repositories.

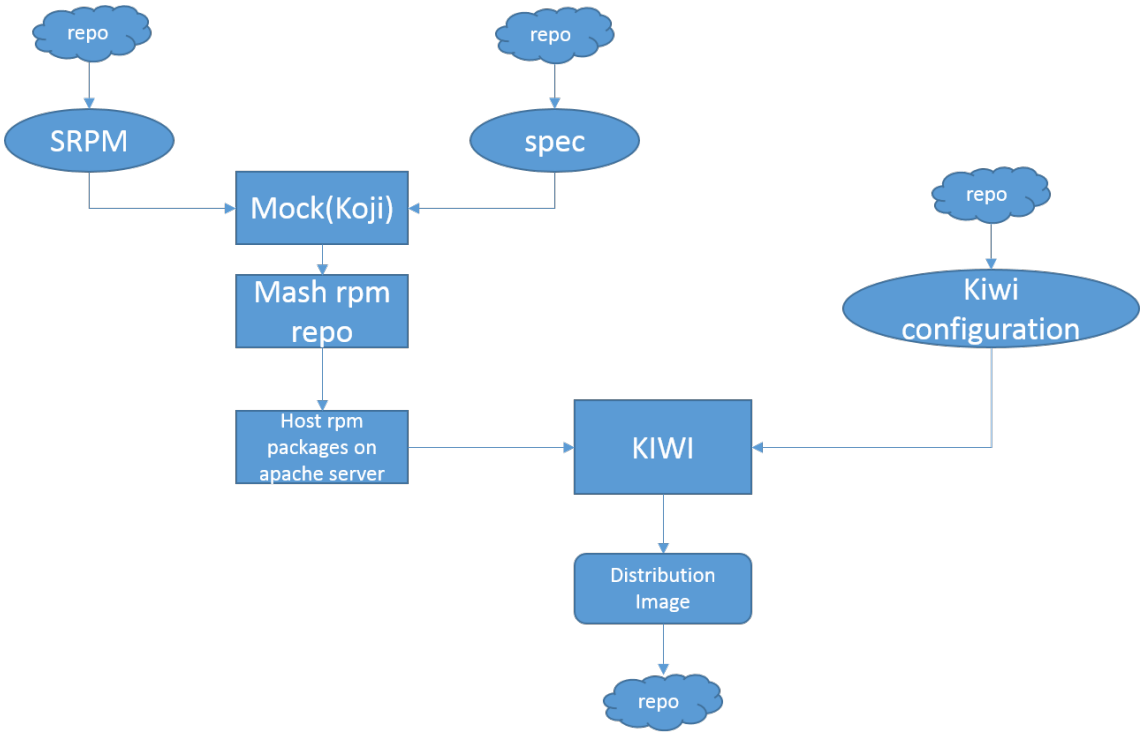

Figure 12: PoC for Process 1, test 3

6. Use Kiwi to build a new distribution from the stored RPM packages and the Kiwi configurations.

7.4.4 Testing the creation process using only Koji's tools

The fourth and the last test for the creation process was conducted with using Koji's tools only. We have an internal RPM repository that is a clone of the Fedora 25 repositories. For this test we used this repository as well as a few packages created with Koji. In this test, illustrated in Figure 13, we first used Koji to build some packages from sources. We did not create a new repository as Koji itself stores the packages built with the Koji's Mock into its own internal cache. This internal cache was used as a source of some of the packages. After this we used the Koji's image building capabilities along with a custom made kickstart file to create a new Linux distribution image. However, due to the lack of time, this image was not tested in the automated testing environment at all. However, as the packages used in this and Nokia Linux distribution images are purely open source, we argue that in the end we would have been successful in pushing this image through our testing environment. We see this only as a matter of creating a configuration that creates a matching

distribution image.

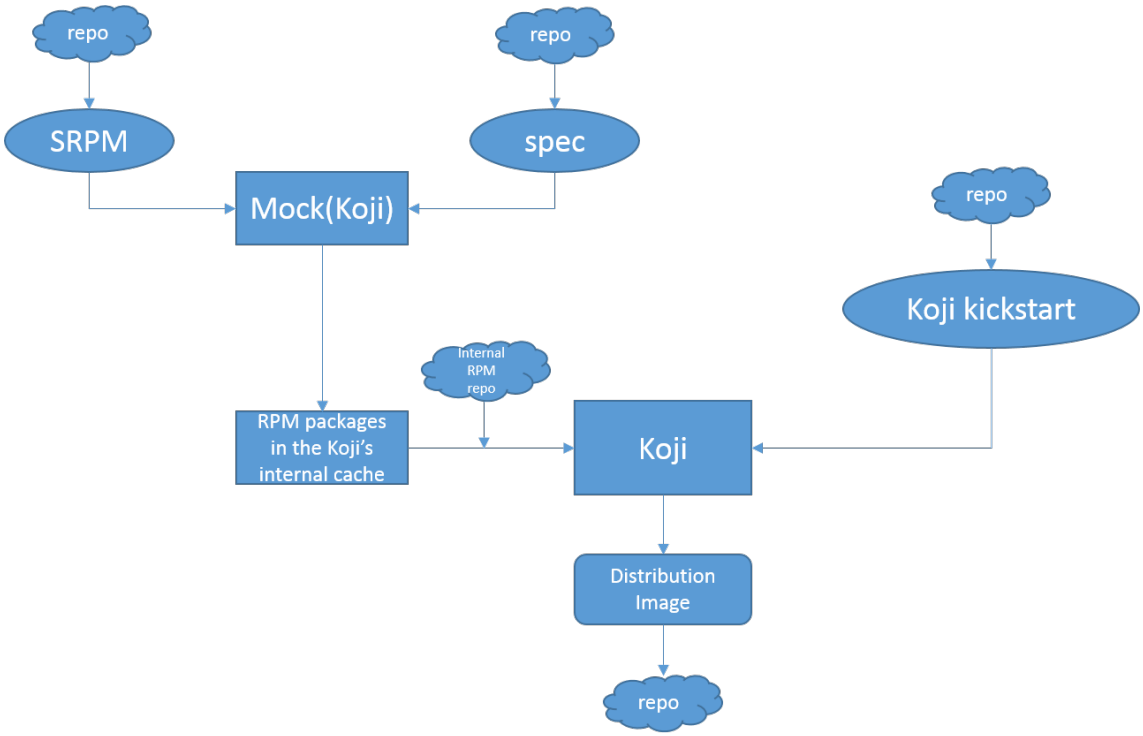

Figure 13: PoC for Process 1, test 4

For this test we created some of the packages using Mock and stored them using Mash like in the third test but instead of using the local repository, we used our internal clone of the Fedora 25 repositories. We then created a Koji configuration to replace the Kiwi configuration. This test was tested using these steps:

1. Acquire sources and acquire or create .spec files for all packages.
2. Using the sources and .spec files, create RPM packages using Mock.
3. Store the newly made RPM packages, the sources and .spec files to a local RPM package repository. Repositories are created using Mash.
4. Create a new Koji configurations for Kiwi.
5. Store Koji configurations to a git repository.
6. Use Koji to build a new distribution from the stored RPM packages and the Kiwi configurations.

7.4.5 Testing the update process using Mock, Mash and Kiwi

The fifth test we conducted tested the update process. We had previously compiled some of our proprietary packages and stored them to the RPM package repository created by Mash. We first installed one of those proprietary packages to our distribution image. We then updated the package, recompiled it and stored the updated package using Mock and Mash. Then we used Kiwi to rebuild the distribution with the updated package. We also tested to have the package manager inside the distribution to update the package.

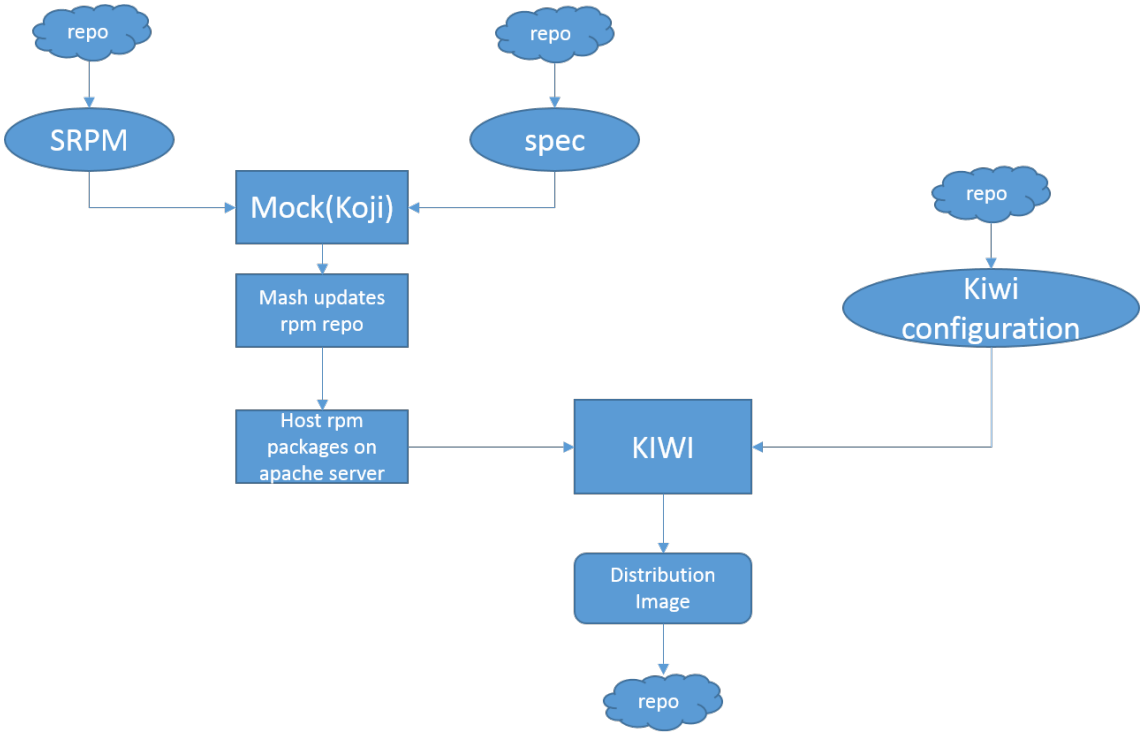

Figure 14: PoC for Process 2

In this test we used the update process to simulate a maintenance of a single package. For the build systems, we used Mock to build the package and Mash to update the repositories. Steps of this test are as follows:

1. For all the RPM packages involved in the maintenance, modify .spec files if necessary and apply updated sources.

2. Using the updated sources and .spec files, recreate updated RPM packages. We used Mock to build the updated package

3. Add the new versions of the RPM packages, their sources and .spec files to the same repository as the old ones keeping the old versions. Mash was used to store the updated package.

4. Modify the Kiwi configuration to use the correct version of the package.

5. Store the updated Kiwi configurations in git.

6. Using the updated package repository and Kiwi configurations recreate the distribution.

7.5 Results of the proof of concept project

All of the tests were successful in their primary objective, which was to show that the different configurations of tools were compliant to our corporate requirements and with our custom software packages. We were able to build or maintain working images using the tools and processes described respectively in each of the test scenarios. However, none of the first four tests were able to fully pass the second phase of testing. We inputted the resulting images into our Jenkins environment to have our product be ran on top. We were unsuccessful in running the image through the entire pipeline.However, this was seen as a secondary objective and we were still able to pass through several phases of the pipeline. Main problems we had were related to gcc and Linux kernel induced errors. All-in-all, we ran the various creation tests dozens of times. Unfortunately, we did not keep track of the actual number of these tests but there were at least 23 test versions in our automation tools at the end of the project. Some versions had been updated many times to fix small errors so we cannot trace back the actual number of the versions. We believe that it is at the very least three times the number.

The first three tests were capable of creating working Linux distributions capable of running at least some components of our products on top of them. However, as stated before, none were able to fully run the product we used for testing on top of them. Main reasons for the failures were version mismatches or system configuration errors. The system configuration errors, such as missing users or permissions, we were able to fix. Most of the components were able to be compiled on top of our new distributions.

In the fourth test, we were successful in creating a new distribution using Koji and a kickstart file. However, this image was not tested in our automated testing

environment because we ran out of time. This image was built using similar list of packages but just with a different Linux build system.

With the fifth test, we successfully created an updated package and then applied it to the existing package repository. We then updated the configuration and ran the build again successfully. We then verified that the package was functioning as expected. In this test, we achieved the secondary objective.

The build times for all of these tests were between 15 and 25 minutes for Kiwi and between 20 and 25 minutes for Koji. The amount of packages seemed to have a significant impact for the build times. A minimalistic image build time for Kiwi was between 7 to 10 minutes, depending on if the build was executed on a virtual machine or on a normal computer. For Koji, similar test took only 5 minutes. For comparison, an average Hammer build takes over one and a half hours. Bear in mind though that Hammer builds all the source packages every time the build is made. Our times with Kiwi and Koji only consists of the build times with ready RPM packages. However, for most of the time, we expect to be modifying only a few packages at a time. This means that, as claimed already before, Hammer is wasting a lot of time.

To summarise, we were successful in showing the usage of the tested processes. However, we were not able to have those images pass the automated test chain completely. The errors we had were mainly version errors but some system configuration errors were also present. These system configuration errors were fixed. The update process was successful in its secondary objective as well. We installed a component of our product on the distribution and then updated it. We then verified that the component still works and this was successful.

8 Discussion

We were partially successful in answering the research questions we had set for this thesis. We first defined some methods for organising support for our distribution. We then defined the processes these support teams could use to create and maintain our own Linux distributions in a more flexible manner than it is currently done. However, as seen from our proof of concept project, we were not completely successful in replicating our current Nokia Linuxes capabilities. We were able to create an image that was capable of going through several of the tests that are ran on the Nokia Linux, but our image was not capable of passing all of them. Therefore we cannot call the RQ3 completely answered yet. The processes failed to fully replicate the Nokia Linux.

However, we feel that the partial failure of the proof of concept project does not completely invalidate our processes. We argue that the failures during the proof of concept project (the version and system configuration errors) were not due to the tools or the processes, but due to the configurations used. For example, Koji is used by Fedora to build Fedora distributions. It already is established that it works. We feel that as none of us had experience with any of the Linux distribution build tools used in this thesis beforehand, the failures of our proof of concept project were caused by our inexperience rather than an error in our processes. With enough time, we argue that we would have been successful in creating working configurations for both Kiwi and Koji.

We did observe a significant speedup when using the Kiwi and Koji build systems over using the Hammer build system. Hammer used almost four times more time in its build than Kiwi or Koji. This is due to one of the major faults of Hammer, the way it builds its packages. As Hammer builds all its packages every time a new distribution is being built, it wastes a lot of time. By using binary packages we save the time required for building most of the packages as we can selectively build only the ones that have changed. Both the Kiwi and Koji build systems would require one complete building of all the used packages and then just updates to a few packages at a time. This saves time over Hammer as Hammer would have to rebuild everything again and again. The savings in time would be huge if Kiwi or Koji would be used over Hammer if suitable configuration files and packages could be found.

8.1 Related work

We used as our main search engines IEEE Xplore, Google Scholar and ACM digital library to search for similar works as our thesis and to see how this kind of processes were designed in the past.

Related works include the Yocto, Koji and kiwi build systems documentations [KoP][YOC][KIS]. All of these projects are about a build system capable of creating a new Linux distribution. As such, they are describing basically the same things that we are describing in this thesis: How to create a new Linux distribution from either sources (Yocto and Koji) or from packages (Kiwi). What we do differently is the inclusion of the support element. Of course, these manual also can and should be used when addressing the problem of how to maintain a system using these build systems but all of them are more technical and describing the systems themselves. We describe in detail how to upgrade packages and what the steps of the support teams should take when not only creating new images but also when maintaining the said images.

Related to our package management, Parker and Ritchey "Automating rpm creation from a source code repository" made a report [PR12] in which they tell about the possibility of using Subversion source code management tool and some custom scripts for creating an automatic RPM build system. This is quite similar on what we had to do with our RPM building system. We used rpmbuild and createrepo tools to build and store the RPM packages from source code in the similar manner. We also had some custom scripts in place.

Adams et al. in their articles [ADSTDM08][ATDSDM07] discuss about how the Linux build systems in general have evolved over time. In other words, how the changes in the source code of Linux has changed the build system of Linux. Adams et al. argue that if the source code changes, the build system must also evolve to match [ADSTDM08]. This relates to our study by giving incentives on why to even do this kind of study in the first place. If our build system would not evolve, the adoption of new software could become impractical.

8.2 Use of the processes outside of Nokia

Assuming that we would be successful in creating the required configurations, the processes defined in this thesis would produce a more flexible and faster build system capable of building new Linux distributions from sources than the Hammer build

system. These improvements should also be observable when comparing to other Linux-from-scratch build systems. More study would be required to verify this however. These processes are general so that the tools used in different steps of the process need not be aware of each other. The only thing required is that the output of the step before must be usable of the step below. This means that these processes could be utilised by others outside of Nokia as there is no vendor lock in with the tools used.

8.3 Further studies

Further studies for the usability of the support models should be made. We presented three models and none were tested in practise. We believe that doing a proof of concept project and actual field trial of the support models would yield some results showing the capabilities of each of the models. We did not test the third process so that is also something to study more.

Build system testing. Testing of the build systems was not studied in this thesis at all, and a separate study should be made. By testing of the build systems we mean the verifying of the validity of the build systems. We need to make sure that the builds are reproducible and error free. For packages, this could be done like Debian is doing it: bit by bit [DeR]. Another possibility would be to check for example the metadata of each package and check if those would be correct. The latter of these would be suitable for RPM packages.

Versioning of the distributions and build system configurations. As the new distributions should be recreatable using the same build system and the same build system configuration, the new distribution needs to have some way to identify with which build system configuration it was created with.

Building docker container distribution images. We very briefly tested our build system and processes to see if docker containers would be possible to be built. At least Kiwi was used to build docker base image. However, no large scale testing was conducted and certainly more study could be done with this approach.

9 Conclusions

Our three corporate environmental needs are not unique. The first environmental need, support, has been monetised by many open source companies such as Red Hat

and Zimbra. The other two, security and storage, can be implemented by using the Koji tool or a toolset of rpmbuild, createrepo and Kiwi, to name a few. As Koji is extensively used by the Fedora project we can argue that security and storage have been implemented already by the Fedora project. This means that the processes and methods we describe in this thesis have already been most likely invented and are being used by other companies and projects already.

However, as we were unsuccessful in finding a similar work to ours, we argue that this particular set of corporate environments is not well tested, documented or made public. Corporate requirements might dictate that the solutions to the requirements be kept secret.

What we did differently in our thesis project, we combined all of the solutions to these three corporate requirements to a suite of processes. This suite of processes would then be used by a support team. We presented different options that could work not only in our corporate environment but in other corporate environments as well. Our contribution is mainly to provide a case study and an example on how to use the build systems with these particular corporate environmental requirements.

In our proof of concept project we were unable to test the support methods due to our tight schedule and lack of other resources. On the other hand, even though we were not quite successful in creating a proper configuration that would be similar to the current Nokia Linux, we were able to show our processes in action in our corporate environment. This was the main objective of the thesis. Despite the fact that we were not able to successfully test any of of our distributions in our automated testing environment, we still managed to show the three processes to be able to perform as they were described. We were completely successful in creating and updating a distribution in our corporate environment. What we were lacking was the functional parity with our Nokia Linux. We still strongly believe that this is achievable, given enough time.

What we can learn from this thesis project is that there seems to be no build system that works with the package versions we need out of the box. Although Kiwi was easy to install and even relatively easy to configure to produce a working, testable image, we required the setting up of our own package build system and package repositories. Koji was hard to install but, similar to Kiwi, relatively easy to configure to produce a working, testable image. Unlike Kiwi, Koji provided a set of RPM package and repository build tools.

With both of these tool sets, we could build our own packages from purpose tailored

.spec files and sources, we still could not complete the testing of the images. This was due to the significant amount of packages that we had to implement. Again, we did simply not have enough time allocated to build most of the packages required.

The packages used were carefully selected to not cause dependency problems. When tested with just a single package taken outside, problems without trivial solutions occurred, mostly due to version mismatches. These package problems could have been described as being a certain kind of dependency hell, the circular dependencies. This was tested with taking some of the packages from the openSUSE repositories and some from the Fedora repositories. The reason why we wanted to take packages from different repositories was because we did not find any single repository with all the required packages present. Our work on creating packages continues after this thesis project to ensure we can use the processes to create a distribution with functional parity with the Nokia Linux.

Time played a large role in this thesis. It took a couple of months to find the focus for the work. We had only a very limited time frame and taking months of this prevented the testing of the third process as well as the finalisation of the configuration files. The limitations of the Hammer build system could be fixed by chopping the build process of the Hammer into smaller pieces like we did with our processes. This would enable us to save time from the users of Nokia Linux. However, as we were only partially successful in testing our processes, we propose a continuation of this thesis. With more time we argue that we could finalise the configurations of our build systems and be successful in discovering a new way of building Nokia's Linux distributions.

10 Acknowledgements

First of all, I want to express my utmost gratitude to Jan Zizka, for supporting me in every possible way in the creation of this thesis. His expertise on the inner workings of the Nokia Linux has been invaluable in the creation of this work. He helped me greatly to find the focus on my thesis and to enable me to go forward with the project.

I want to thank Tomi Männistö and Tommi Mikkonen for providing me with critical ideas that helped greatly widen my understanding on the subject of design science and most certainly improve my scientific writing.

I want to thank Nokia for providing me with this excellent opportunity. Without the support of Nokia, I could have not made this thesis.

Last but most certainly not the least I want to thank my coworkers Samu Toimela, Jani Kuutvuori, Miro Äijälä, Toni Kangas and Niko Kortström for their constructive comments, criticism and eagle eyed proof reading of my thesis. They provided me with a working environment par excellence in which I was able to do research in an innovative and creative way.

References

ADSTDM08 Adams, B., De Schutter, K., Tromp, H. and De Meuter, W., The evolution of the linux build system. 2008, *Electronic Communications of the EASST*, 8.

AKH Hammer build system, `https://github.com/aakoskin/hammer`. Accessed: 26.9.2016.

ASDC+12 Artho, C., Suzaki, K., Di Cosmo, R., Treinen, R. and Zacchiroli, S., Why Do Software Packages Conflict? *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, Piscataway, NJ, USA, 2012, IEEE Press, pages 141–150, URL `http://dl.acm.org/citation.cfm?id=2664446.2664470`.

ATDSDM07 Adams, B., Tromp, H., De Schutter, K. and De Meuter, W., Design recovery and maintenance of build systems. *2007 IEEE International Conference on Software Maintenance*. IEEE, 2007, pages 114–123.

CHR Chroot man pages, `https://linux.die.net/man/2/chroot`. Accessed: 2.3.2017.

CLK A Closer Look at the KIWI Imaging System, `https://www.linux.com/news/closer-look-kiwi-imaging-system`. Accessed: 10.11.2016.

CQS Code Quality Standards, `http://www.it-cisq.org/standards/`. Accessed: 30.11.2016.

CRM createrepo man pages, `https://linux.die.net/man/8/createrepo`. Accessed: 2.3.2017.

Deb Debian, `https://www.debian.org/`. Accessed: 26.9.2016.

DeR Debian reproducability, `https://wiki.debian.org/ReproducibleBuilds/About`. Accessed: 3.2.2017.

DG01 Dinkelacker, J. and Garg, P., Corporate source: Applying open source concepts to a corporate environment (position paper). 2001, *1st WS Open Source SE*.

DIB Disk image builder wiki, `http://docs.openstack.org/developer/diskimage-builder/`. Accessed: 17.1.2017.

DT03 Davidson, E. J. and Tay, A. S., Studying teamwork in global IT support. *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*. IEEE, 2003.

ESF Expect spec file, `https://build.opensuse.org/package/view_file/SUSE:SLE-12-SP1:GA/expect/expect.spec?expand=1`. Accessed: 6.3.2017.

FCP Fedora cloud base kickstart, `https://pagure.io/fedora-kickstarts/blob/master/f/fedora-cloud-base.ks`. Accessed: 2.3.2017.

FeP Fedora Project, `https://fedoraproject.org/wiki/Fedora_Project_Wiki`. Accessed: 26.9.2016.

GRT Gartner, Worldwide Smartphone Sales to End Users by Operating System in 2Q16 (Thousands of Units), `http://www.gartner.com/newsroom/id/3415117`. Accessed: 18.11.2016.

HaB Heartbleed, `http://heartbleed.com/`. Accessed: 26.9.2016.

HD02 Hart, J. and D'Amelia, J., An Analysis of RPM Validation Drift. *LISA*, volume 2, 2002, pages 155–166.

HFKM14 Hieber, P., Feldmaier, J., Knopp, M. and Meyer, D., Yocto project on the gumstix overo board. 2014, *Technische Universitt Munchen*.

JKK Jensen, C., King, S. and Kuechler, V., Joining free/open source software communities: An analysis of newbies' first interactions on project mailing lists.

KIS Kiwi Image system, `https://doc.opensuse.org/projects/kiwi/doc/`. Accessed: 10.11.2016.

KoP Koji Project, `https://fedoraproject.org/wiki/Koji`. Accessed: 10.11.2016.

Kro Krob, A., Finding a Balance: Centralized IT Support for Decentralized Units and the Liaison Program at Tulane University. *Proceedings of the 34th Annual ACM SIGUCCS Fall Conference: Expanding the Boundaries*, SIGUCCS '06, year = 2006, isbn = 1-59593-438-3, location =.

KSe Setting up a Koji Build System, https://docs.pagure.org/koji/server_howto/. Accessed: 10.11.2016.

Kua02 Kuan, J., Open source software as lead user's make or buy decision: a study of open and closed source quality. 2002, *Stanford Institute for Economic Policy Research, Stanford University*.

LEm Embedded Linux Keeps Growing Amid IoT Disruption, Says Study, https://www.linux.com/news/embedded-linux-keeps-growing-amid-iot-disruption-says-study. Accessed: 29.11.2016.

Lin Linux communtiy guidelines, https://www.linux.com/publications/how-participate-linux-community. Accessed: 26.4.2017.

MCK rpm Mock wiki, https://github.com/rpm-software-management/mock/wiki. Accessed: 2.3.2017.

MSH Mash pagure wiki, https://pagure.io/mash. Accessed: 2.3.2017.

OPR openSUSE repositories, https://en.opensuse.org/Package_repositories. Accessed: 3.2.2017.

PR12 Parker, T. and Ritchey, P., Automating RPM Creation from a Source Code Repository. Technical Report, DTIC Document, 2012.

Ray99 Raymond, E., The cathedral and the bazaar. 1999, *Knowledge, Technology & Policy*, 12,3(1999), pages 23–49.

RBM rpmbuild man pages, https://linux.die.net/man/8/rpmbuild. Accessed: 2.3.2017.

RHK Red Hat kickstart manual, https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Installation_Guide/s1-kickstart2-packageselection.html. Accessed: 6.3.2017.

RHM Red Hat manual, https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/System_Administrators_Guide/sec-Working_with_Package_Groups.html. Accessed: 2.3.2017.

RHS Red Hat support portal, `https://access.redhat.com/`. Accessed: 2.3.2017.

SB03 Sneed, H. M. and Brossler, P., Critical success factors in software maintenance: a case study. *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 2003, pages 190–198.

Sch11 Schryen, G., Is Open Source Security a Myth? May 2011, *Commun. ACM*, 54,5(2011), pages 130–140. URL `http://doi.acm.org/10.1145/1941487.1941516`.

SGK+09 Spinellis, D., Gousios, G., Karakoidas, V., Louridas, P., Adams, P. J., Samoladas, I. and Stamelos, I., Evaluating the quality of open source software. 2009, *Electronic Notes in Theoretical Computer Science*, 233, pages 5–28.

Spi06 Spinellis, D., *Code quality: the open source perspective*. Adobe Press, 2006.

SUS SUSE Studio, `https://susestudio.com/`. Accessed: 26.9.2016.

TOP TOP 500 The List., `https://www.top500.org/statistics/overtime/`. Accessed: 18.11.2016.

TSHO09 Thomas, L., Schach, S. R., Heller, G. Z. and Offutt, J., Impact of release intervals on empirical research into software evolution, with application to the maintainability of Linux. 2009, *IET software*, 3,1(2009), pages 58–66.

Ubu Ubuntu, `https://wiki.ubuntu.com/`. Accessed: 26.9.2016.

VAMPR04 Von Alan, R. H., March, S. T., Park, J. and Ram, S., Design science in information systems research. 2004, *MIS quarterly*, 28,1(2004), pages 75–105.

WG05 Woods, D. and Guliani, G., *Open Source for the Enterprise: Managing Risks Reaping Rewards*. O'Reilly Media, Inc., 2005.

Win01 Windley, P. J., Delivering high availability services using a multi-tiered support model. 2001, *Utah dot gov*.

YMM Yocto Project Mega-Manual, http://www.yoctoproject.org/docs/2.2/mega-manual/mega-manual.html. Accessed: 21.11.2016.

YOC Yocto Project, https://www.yoctoproject.org. Accessed: 18.11.2016.

YQG Yocto Project Quick start guide, https://www.yoctoproject.org/docs/1.8/yocto-project-qs/yocto-project-qs.html. Accessed: 3.2.2017.

YVB Deciding between Buildroot & Yocto, https://lwn.net/Articles/682540/. Accessed: 18.11.2016.

ZSP Zimbra support portal, https://www.zimbra.com/support/support-offerings/. Accessed: 6.3.2017.

A expect .spec

Source: build.opensuse.org [ESF].

```
#
# spec file for package expect
#
# Copyright (c) 2015 SUSE LINUX GmbH, Nuernberg, Germany.
#
# All modifications and additions to the file contributed by third parties
# remain the property of their copyright owners, unless otherwise agreed
# upon. The license for this file, and modifications and additions to the
# file, is the same license as for the pristine package itself (unless the
# license for the pristine package is not an Open Source License, in which
# case the license is the MIT License). An "Open Source License" is a
# license that conforms to the Open Source Definition (Version 1.9)
# published by the Open Source Initiative.

# Please submit bugfixes or comments via http://bugs.opensuse.org/
#


Url:            http://expect.nist.gov

Name:           expect
BuildRequires:  autoconf
BuildRequires:  tcl-devel
Version:        5.45
Release:        0
BuildRoot:      %{_tmppath}/%{name}-%{version}-build
Summary:        A Tool for Automating Interactive Programs
License:        SUSE-Public-Domain
Group:          Development/Languages/Tcl
Source:         http://downloads.sourceforge.net/project/ \
                %{name}/Expect/%{version}/%{name}%{version}.tar.gz
Source1:        expect-rpmlintrc
Patch1:         expect.patch
```

```
Patch2:         expect-fixes.patch
Patch3:         expect-log.patch
Patch4:         config-guess-sub-update.patch

%description
Expect is a tool primarily for automating interactive applications,
such as telnet, ftp, passwd, fsck, rlogin, tip, and more.  Expect
really makes this stuff trivial.  Expect is also useful for testing
these applications.  It is described in many books, articles, papers,
and FAQs.  There is an entire book on it available from O'Reilly.

%package devel
Summary:        Header Files and C API Documentation for expect
Group:          Development/Libraries/Tcl

%description devel
This package contains header files and documentation needed for linking
to expect from programs written in compiled languages like C, C++, etc.

This package is not needed for developing scripts that run under the
/usr/bin/expect interpreter, or any other Tcl interpreter with the
expect package loaded.

%prep
%setup -q -n %name%version
%patch1
%patch2
%patch3
%patch4

%build
autoreconf
%configure \
--with-tcl=%_libdir \
--with-tk=no_tk \
--with-tclinclude=%_includedir \
```

```
--enable-shared
make %{?_smp_mflags} all pkglibdir=%_libdir/tcl/%name%version

%check
make test

%install
# set the right path to the expect binary...
sed -i \
    -e '1s,^#![^ ]*expectk,#!/usr/bin/wish\npackage require Expect,' \
    -e '1s,^#![^ ]*expect,#!/usr/bin/expect,' \
    example/*
make install DESTDIR=$RPM_BUILD_ROOT pkglibdir=%_libdir/tcl/%name%version
# Remove some executables and manpages we don't want to ship
rm $RPM_BUILD_ROOT%_prefix/bin/*passwd
rm $RPM_BUILD_ROOT%_prefix/bin/weather
rm $RPM_BUILD_ROOT%_mandir/*/*passwd*

%files
%defattr(-,root,root)
%_prefix/bin/*
%_libdir/tcl/*
%_libdir/lib*.so
%doc %_mandir/man1/*
%doc ChangeLog HISTORY INSTALL FAQ NEWS README

%files devel
%defattr(-,root,root)
%_includedir/*
%doc %_mandir/man3/*

%changelog
```

B Kiwi config.xml

This file was pre-generated by SUSEStudio and then later modified.

```
<?xml version='1.0' encoding='UTF-8'?>
<image name='poc' displayname='poc' schemaversion='5.2'>
  <description type='system'>
    <author>Juhani H</author>
    <contact>studio-devel@suse.de</contact>
    <specification>Tiny, minimalistic appliances</specification>
  </description>
  <preferences>
    <type checkprebuilt='true' boot='vmxboot/suse-13.2' fsnocheck='true'
    filesystem='ext3' bootloader='grub2' format='qcow2'
    kernelcmdline='console=tty console=ttyS0,38400' fsmountoptions='acl'
    image='vmx'>
      <size unit='M' additive='false'>16384</size>
    </type>
    <version>0.0.1</version>
    <packagemanager>zypper</packagemanager>
    <rpm-check-signatures>false</rpm-check-signatures>
    <rpm-force>false</rpm-force>
    <boot-theme>studio</boot-theme>
    <timezone>UTC</timezone>
    <hwclock>localtime</hwclock>
  </preferences>
  <users group='root'>
    <-- users -->
  </users>
  <packages type='image' patternType='onlyRequired'>
    <-- distribution packages -->
  </packages>
  <packages type='image' patternType='onlyRequired'>
    <-- devel packages -->
  </packages>
  <packages type='bootstrap'>
    <package name='filesystem'/>
    <package name='glibc-locale'/>
    <package name='module-init-tools'/>
  </packages>
```

```
  <repository type='rpm-md'>
    <source path='/root/repos/localrepo/'/>
  </repository>
  <repository type='rpm-md'>
    <source path='/root/repos/localupdates/'/>
  </repository>
</image>
```

C Kiwi config.sh

This file was pre-generated by SUSEStudio and then later modified.

```
#!/bin/bash
#================
# FILE          : config.sh
#----------------
# PROJECT       : OpenSuSE KIWI Image System
# COPYRIGHT     : (c) 2006 SUSE LINUX Products GmbH. All rights reserved
#               :
# AUTHOR        : Marcus Schaefer <ms@suse.de>
#               :
# BELONGS TO    : Operating System images
#               :
# DESCRIPTION   : configuration script for SUSE based
#               : operating systems
#               :
#               :
# STATUS        : BETA
#----------------
#=======================================
# Functions...
#---------------------------------------
test -f /.kconfig && . /.kconfig
test -f /.profile && . /.profile
```

```
#======================================
# Greeting...
#--------------------------------------
echo "Configure image: [$name]..."

#======================================
# SuSEconfig
#--------------------------------------
echo "** Running suseConfig..."
suseConfig

echo "** Running ldconfig..."
/sbin/ldconfig

#======================================
# Setup default runlevel
#--------------------------------------
baseSetRunlevel 3

#======================================
# Add missing gpg keys to rpm
#--------------------------------------
suseImportBuildKey


sed --in-place -e 's/# solver.onlyRequires.*/solver.onlyRequires = true/' \
/etc/zypp/zypp.conf

# Enable sshd
chkconfig sshd on

#======================================
# Sysconfig Update
#--------------------------------------
echo '** Update sysconfig entries...'
baseUpdateSysConfig /etc/sysconfig/keyboard KEYTABLE us.map.gz
```

```
baseUpdateSysConfig /etc/sysconfig/network/config FIREWALL no
baseUpdateSysConfig /etc/sysconfig/network/config LINKLOCAL_INTERFACES ''
baseUpdateSysConfig /etc/init.d/suse_studio_firstboot NETWORKMANAGER no
baseUpdateSysConfig /etc/sysconfig/SuSEfirewall2 FW_SERVICES_EXT_TCP 22\ 80\ 443
baseUpdateSysConfig /etc/sysconfig/console CONSOLE_FONT lat9w-16.psfu

(Some Nokia specific scripts)

#installed pip is very old, upgrade
pip install --upgrade pip

(Some Nokia specific scripts)

test -d /studio || mkdir /studio
cp /image/.profile /studio/profile
cp /image/config.xml /studio/config.xml
chown root:root /studio/build-custom
chmod 755 /studio/build-custom
# run custom build_script after build
if ! /studio/build-custom; then
    cat <<EOF

*********************************
/studio/build-custom failed!
*********************************

EOF

    exit 1
fi
rm -rf /studio/overlay-tmp
true

#====================================
# SSL Certificates Configuration
#------------------------------------
```

```
echo '** Rehashing SSL Certificates...'
c_rehash
```

D Koji kickstart

This is built on top the fedora-cloud-base.ks from fedoras own repository [FCP].

```
# This is a basic Fedora 21 spin designed to work in OpenStack and other
# private cloud environments. It's configured with cloud-init so it will
# take advantage of ec2-compatible metadata services for provisioning ssh
# keys. Cloud-init creates a user account named "fedora" with passwordless
# sudo access. The root password is empty and locked by default.
#
# Note that unlike the standard F20 install, this image has /tmp on disk
# rather than in tmpfs, since memory is usually at a premium.
#
# This kickstart file is designed to be used with ImageFactory (in Koji).
#
# To do a local build, you'll need to install ImageFactory.  See
# http://worknotes.readthedocs.org/en/latest/cloudimages.html for some notes.
#
# For a TDL file, I store one here:
# https://git.fedorahosted.org/cgit/fedora-atomic.git/tree/
# fedora-atomic-rawhide.tdl
# (Koji generates one internally...what we really want is Koji to
# publish it statically)
#
# Once you have imagefactory and imagefactory-plugins installed, run:
#
#   curl -O https://git.fedorahosted.org/cgit/fedora-atomic.git/plain/ \
#   fedora-atomic-rawhide.tdl
#   tempfile=$(mktemp --suffix=.ks)
#   ksflatten -v F22 -c fedora-cloud-base.ks > ${tempfile}
#   imagefactory --debug base_image --file-parameter install_script \
#   ${tempfile} fedora-atomic-rawhide.tdl
```

```
#

text # don't use cmdline -- https://github.com/rhinstaller/anaconda/issues/931
lang en_US.UTF-8
keyboard us
timezone --utc Etc/UTC

auth --useshadow --passalgo=sha512
selinux --enforcing
user --name=none

firewall --disabled

bootloader --timeout=1 --append="no_timer_check console=tty1 \
console=ttyS0,115200n8" --extlinux

network --bootproto=dhcp --device=link --activate --onboot=on
services --enabled=sshd,cloud-init,cloud-init-local,cloud-config,cloud-final

zerombr
clearpart --all
#
# We need to disable 64bit options here or extlinux won't work.
# See: http://www.syslinux.org/wiki/index.php/Filesystem#ext4
# and
# https://bugzilla.redhat.com/show_bug.cgi?id=1369934
#
part / --fstype ext4 --grow --mkfsoptions="-O ^64bit"

%include /home_local/kojiadmin/fedora-repo.ks

reboot

# Package list.
# FIXME: instLangs does not work, so there's a hack below
# (see https://bugzilla.redhat.com/show_bug.cgi?id=1051816)
```

```
# FIXME: instLangs bug has been fixed but now having instLangs
# with an arg causes no langs to get installed because of BZ1262040
# which yields the errors in BZ1261249. For now fix by not using
# --instLangs at all
#%packages --instLangs=en
%packages

kernel-core
@^cloud-server-environment
# Need to pull in the udev subpackage
systemd-udev

# after move away from grub2 - let's add 'which' back
which

# rescue mode generally isn't useful in the cloud context
-dracut-config-rescue

# Some things from @core we can do without in a minimal install
-biosdevname
# Need to also add back plymouth in order to mask failure of
# systemd-vconsole-setup.service. BZ#1272684. Comment out for now
#-plymouth
-NetworkManager
-iprutils
# Now that BZ#1199868 is fixed kbd really gets removed but it breaks
# systemd-vconsole-setup.service on boot. Comment out for now
#-kbd
-uboot-tools
-kernel
-grub2

#---------------- Our packages below this line ------------------

(our installed list of packages)
```

```
#---------------- Our packages above this line ------------------


%end



%post --erroronfail

# Create grub.conf for EC2. This used to be done by appliance creator but
# anaconda doesn't do it. And, in case appliance-creator is used, we're
# overriding it here so that both cases get the exact same file.
# Note that the console line is different -- that's because EC2 provides
# different virtual hardware, and this is a convenient way to act differently
echo -n "Creating grub.conf for pvgrub"
rootuuid=$( awk '$2=="/" { print $1 };'  /etc/fstab )
mkdir /boot/grub
echo -e 'default=0\ntimeout=0\n\n' > /boot/grub/grub.conf
for kv in $( ls -1v /boot/vmlinuz* |grep -v rescue |sed s/.*vmlinuz-//  ); do
  echo "title Fedora ($kv)" >> /boot/grub/grub.conf
  echo -e "\troot (hd0,0)" >> /boot/grub/grub.conf
  echo -e "\tkernel /boot/vmlinuz-$kv ro root=$rootuuid no_timer_check \
  console=hvc0 LANG=en_US.UTF-8" >> /boot/grub/grub.conf
  echo -e "\tinitrd /boot/initramfs-$kv.img" >> /boot/grub/grub.conf
  echo
done


#link grub.conf to menu.lst for ec2 to work
echo -n "Linking menu.lst to old-style grub.conf for pv-grub"
ln -sf grub.conf /boot/grub/menu.lst
ln -sf /boot/grub/grub.conf /etc/grub.conf

# older versions of livecd-tools do not follow "rootpw --lock" line above
# https://bugzilla.redhat.com/show_bug.cgi?id=964299
passwd -l root
```

```
# remove the user anaconda forces us to make
userdel -r none

# Kickstart specifies timeout in seconds; syslinux uses 10ths.
# 0 means wait forever, so instead we'll go with 1.
sed -i 's/^timeout 10/timeout 1/' /boot/extlinux/extlinux.conf

# setup systemd to boot to the right runlevel
echo -n "Setting default runlevel to multiuser text mode"
rm -f /etc/systemd/system/default.target
ln -s /lib/systemd/system/multi-user.target /etc/systemd/system/default.target
echo .

# this is installed by default but we don't need it in virt
# Commenting out the following for #1234504
# rpm works just fine for removing this, no idea why dnf can't cope
echo "Removing linux-firmware package."
rpm -e linux-firmware

# Remove firewalld; was supposed to be optional in F18+, but is pulled in
# in install/image building.
echo "Removing firewalld."
# FIXME! clean_requirements_on_remove is the default with DNF, but may
# not work when package was installed by Anaconda instead of command line.
# Also -- check if this is still even needed with new anaconda -- disabled
# firewall should _not_ pull in this package.
# yum -C -y remove "firewalld*" --setopt="clean_requirements_on_remove=1"
dnf -C -y erase "firewalld*"

# Another one needed at install time but not after that, and it pulls
# in some unneeded deps (like, newt and slang)
echo "Removing authconfig."
dnf -C -y erase authconfig

# instlang hack. (Note! See bug referenced above package list)
find /usr/share/locale -mindepth  1 -maxdepth 1 -type d -not \
```

```
-name en_US -exec rm -rf {} +
localedef --list-archive | grep -v ^en_US | xargs localedef \
--delete-from-archive
# this will kill a live system (since it's memory mapped) but
# should be safe offline
mv -f /usr/lib/locale/locale-archive /usr/lib/locale/locale-archive.tmpl
build-locale-archive
echo '%_install_langs C:en:en_US:en_US.UTF-8' \
>> /etc/rpm/macros.image-language-conf


echo -n "Getty fixes"
# although we want console output going to the serial console, we don't
# actually have the opportunity to login there. FIX.
# we don't really need to auto-spawn _any_ gettys.
sed -i '/^#NAutoVTs=.*/ a\
NAutoVTs=0' /etc/systemd/logind.conf

echo -n "Network fixes"
# initscripts don't like this file to be missing.
# and https://bugzilla.redhat.com/show_bug.cgi?id=1204612
cat > /etc/sysconfig/network << EOF
NETWORKING=yes
NOZEROCONF=yes
DEVTIMEOUT=10
EOF

# For cloud images, 'eth0' _is_ the predictable device name, since
# we don't want to be tied to specific virtual (!) hardware
rm -f /etc/udev/rules.d/70*
ln -s /dev/null /etc/udev/rules.d/80-net-setup-link.rules

# simple eth0 config, again not hard-coded to the build hardware
cat > /etc/sysconfig/network-scripts/ifcfg-eth0 << EOF
DEVICE="eth0"
BOOTPROTO="dhcp"
```

```
ONBOOT="yes"
TYPE="Ethernet"
PERSISTENT_DHCLIENT="yes"
EOF

# generic localhost names
cat > /etc/hosts << EOF
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6

EOF
echo .


# Because memory is scarce resource in most cloud/virt environments,
# and because this impedes forensics, we are differing from the Fedora
# default of having /tmp on tmpfs.
echo "Disabling tmpfs for /tmp."
systemctl mask tmp.mount

# make sure firstboot doesn't start
echo "RUN_FIRSTBOOT=NO" > /etc/sysconfig/firstboot

# Uncomment this if you want to use cloud init but suppress the creation
# of an "ec2-user" account. This will, in the absence of further config,
# cause the ssh key from a metadata source to be put in the root account.
#cat <<EOF > /etc/cloud/cloud.cfg.d/50_suppress_ec2-user_use_root.cfg
#users: []
#disable_root: 0
#EOF

echo "Removing random-seed so it's not the same in every image."
rm -f /var/lib/systemd/random-seed

echo "Cleaning old dnf repodata."
# FIXME: clear history?
```

```
dnf clean all
truncate -c -s 0 /var/log/dnf.log
truncate -c -s 0 /var/log/dnf.rpm.log

echo "Import RPM GPG key"
releasever=$(rpm -q --qf '%{version}\n' fedora-release)
basearch=$(uname -i)
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-fedora-$releasever-$basearch

echo "Packages within this cloud image:"
echo "-------------------------------------------------------------------------"
rpm -qa
# Note that running rpm recreates the rpm db files which aren't needed/wanted
rm -f /var/lib/rpm/__db*


# This is a temporary workaround for
# <https://bugzilla.redhat.com/show_bug.cgi?id=1147998>
# where sfdisk seems to be messing up the mbr.
# Long-term fix is to address this in anaconda directly and remove this.
# <https://bugzilla.redhat.com/show_bug.cgi?id=1015931>
dd if=/usr/share/syslinux/mbr.bin of=/dev/vda

# FIXME: is this still needed?
echo "Fixing SELinux contexts."
touch /var/log/cron
touch /var/log/boot.log
chattr -i /boot/extlinux/ldlinux.sys
/usr/sbin/fixfiles -R -a restore
chattr +i /boot/extlinux/ldlinux.sys

echo "Zeroing out empty space."
# This forces the filesystem to reclaim space from deleted files
dd bs=1M if=/dev/zero of=/var/tmp/zeros || :
rm -f /var/tmp/zeros
echo "(Don't worry -- that out-of-space error was expected.)"
```

```
# For trac ticket https://fedorahosted.org/cloud/ticket/128
rm -f /etc/sysconfig/network-scripts/ifcfg-ens3

# Enable network service here, as doing it in the services line
# fails due to RHBZ #1369794
/sbin/chkconfig network on

# Remove machine-id on pre generated images
rm -f /etc/machine-id
touch /etc/machine-id

%end
```

E Repository update script using createrepo

```
#!/bin/bash
echo "update repo"
cd ./localrepo/
createrepo ./
chmod o-w+r ./

sleep 1
echo "update updaterepo"
cd ../localupdates/
createrepo ./
chmod o-w+r ./

echo "done"
```

F Mash config

```
# mash config file

[custom-release]
```

```
rpm_path = %(arch)s/os/Packages
repodata_path = %(arch)s/os/
source_path = source/SRPMS
debuginfo = True
multilib = False
multilib_method = devel
hash = sha
tag = dist-foo
inherit = False
#strict_keys = False
#keys = 81b46521
arches = x86_64
delta = False
hash_packages = False
latest = False
```